PROCEEDINGS OF THE THIRTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

NOVEMBER 1988

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

PROCEEDINGS

OF THE

THIRTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Organized by:

Software Engineering Laboratory
GSFC

November 30, 1988

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development Branch)
The University of Maryland (Computer Sciences Department)
The Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models in the process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained from:

NASA/Goddard Space Flight Center
Systems Development Branch
Code 552
Greenbelt, Maryland 20771

AGENDA

THIRTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 8 AUDITORIUM
NOVEMBER 30, 1988

8:00 a.m.	Registration - 'Sign-In' Coffee, Donuts	
8:45 a.m.	INTRODUCTORY REMARKS	Frank E. McGarry (NASA/GSFC)
9:00 a.m.	Session No. 1	Topic: Studies and Experiments in the SEL
	"Evolving Impacts of Ada on a Production Environment"	Frank McGarry (NASA/GSFC) Linda Esker and Kelvin Quimby (CSC)
	"Measuring/Reusing and Maintaining Ada Software"	Vic Basili and Marv Zelkowitz (Univ. of MD)
	"The Software Management Environment"	Jon Valett (NASA/GSFC) Bill Decker and John Buell (CSC)
10:30 a.m.	BREAK	
11:00 a.m.	Session No. 2	Topic: Software Models Discussant: Jerry Page (CSC)
	"A Communication Channel Model of the Software Process"	Robert Tausworthe (JPL)
	"Knowledge-Based Assistance in Costing the Space Station Data Management System"	Troy Henson and Kyle Rone (IBM)
	"Software Sizing, Cost Estimation and Scheduling"	William Cheadle (Martin Marietta)
12:30 p.m.	LUNCH	

AGENDA (Con't)

1:30 p.m.	Session No. 3	Topic: Study of Software Products Discussant: John Musa (Bell/Labs)
	"Reverse Engineering: An Aid in Understanding"	Hasan Sayani (ASTEC)
	"Ada Software Productivity Analysis"	Jairus M. Hihn, Hamid Habib-Agahi and Shan Malhotra (JPL)
	"Experiences with Ada in an Embedded System"	Robert LaBaugh (Martin Marietta)
3:00 p.m.	BREAK	
3:30 p.m.	Session No. 4	Topic: Tools Discussant: Mike Gardner (CSC)
	"A Practical Approach to Object Based Requirements Analysis"	Daniel W. Drew Michael Bishop (Unisys)
	"A Modernized PDL Approach for Ada Software Development"	Paul Usavage, Jr (GE)
	"Representing Object-Oriented Specifications and Designs with Extended Data Flow Notation"	Jon Franklin Buser and Paul T. Ward (Software Development Concepts)
5:00 p.m.	ADJOURN	

SUMMARY OF THE THIRTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

By

Linda Landis

COMPUTER SCIENCES CORPORATION

SUMMARY OF THE THIRTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

On November 30, 1988, approximately 450 attendees gathered in Building 8 at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) for the Thirteenth Annual Software Engineering Workshop. The meeting is held each year as a forum for information exchange in the measurement, utilization, and evaluation of software methods, models, and tools. It is sponsored by the Software Engineering Laboratory (SEL), a cooperative effort of NASA/GSFC, the University of Maryland, and Computer Sciences Corporation (CSC). Among the audience were representatives from 6 universities, 22 government agencies, 8 NASA centers, and 78 private corporations and institutions. Twelve papers were presented in four sessions:

- Studies and Experiments in the SEL
- Software Models
- Study of Software Products
- Tools

Speakers accepted questions after their presentations and during panel discussions at the end of each session. Responses and comments elicited from audience members resulted in a lively exchange.

SESSION 1 - STUDIES AND EXPERIMENTS IN THE SEL

Frank McGarry of GSFC introduced the workshop and opened the session. In his presentation (Evolving Impacts of Ada on a Production Environment), McGarry addressed five major questions:

- What is the impact of Ada on development profiles?
- What are its effects on productivity, reliability, and maintainability?

L. Landis
CSC

- How does the impact change from first-time Ada use through third-time?
- Do we use Ada differently over time?
- How long does it take to reap the promised benefits of Ada?

McGarry described the use of Ada on NASA/GSFC Flight Dynamics Division (FDD) projects and characterized each project by level of Ada experience. He found that the first Ada project had a phase distribution similar to that of a parallel FORTRAN project for predesign, design, code, and test as a percentage of total effort. The predicted shift to more effort in design did occur on subsequent Ada projects but was not observed on projects characterized as first-time Ada use. Productivity statistics showed that the total lines of code (LOC) per staff day improved significantly from first-time projects to those of third-time Ada use. The trend in number of statements per staff day was also up, although the FORTRAN project's statistic remained higher.

McGarry emphasized that the use of Ada features changed appreciably with experience; the use of generics, strong typing, and packages increased while the use of tasking declined. He also concluded that the use of Ada reduces interface errors. In summary, McGarry noted the following:

- Overhead cost of Ada usage was 30 percent in first-time projects, but significant improvement was noticed in second- and third-time projects.
- Reliability was similar to FORTRAN initially but improved with experience.
- Positive trends in reuse were noted, already exceeding FORTRAN.

- Ada projects have a higher total LOC than FORTRAN projects, but the number of statements is approximately equal.
- The use of Ada features evolved with experience and appears related to improved productivity and reliability, although certain features were found to be inappropriate for the FDD environment.

The second presentation (Toward a Reuse-Oriented Software Evolution Process) was given by Victor Basili of the University of Maryland. The problem Basili posed was that, although reuse of experience is key to productivity and quality, current reuse practices are ad hoc, implicit, and at the code level. Reuse, he stated, must be built into the software development process, and models of the reuse environment must be constructed. Reuse in the traditional, project-specific, SEL software evolution environment is not only explicit through code, Basili found, but is implicit through people; the same processes, management, and support tools have been used by SEL projects over a long period.

Basili proposed a reuse-oriented software evolution model that would supplant the traditional model. It would incorporate improvements in software development by recording learning in a repository of well-classified experience (the experience base). The goal would be to maximize the use of the recorded information during project planning and execution. The experience would be massaged off-line to generalize the information gathered and would be tailored on-line for specific project applications as needed. Formalization would encode the experience in a more precise, understandable manner.

Basili concluded that integrated models are needed for all activities to achieve maximum reuse and minimum tailoring. Models and project goals are also required to develop useful

measures of reuse, as opposed to source LOC (SLOC). In the SEL, the movement to Ada has incurred costs in the short run, but explicit reuse characterization can and does help.

Jon Valett of GSFC presented the third paper of the session (The Software Management Environment (SME)). The goals of the SME project, Valett explained, are to integrate experience and knowledge from completed software projects and feed it back to management. The process is automated via a tool set that uses historical information about software development. The SME compares development profiles of current versus past projects; predicts cost, reliability, and error rates; analyzes the strengths and weaknesses of projects; and provides expert guidance regarding overall project quality.

The SME is constructed in Pascal on a VAX-11/780 computer connected to IBM personal computers (PCs). Its components are the SEL database, models and measures created as a result of SEL research, and software development rules. The SEL database contains data on resource utilization, project growth, and methodology characterization. The rules are based on information obtained from experienced managers and from analyses of collected information and models.

Valett then showed how the SME would respond to a sample question: "How does my project compare with other projects in respect to number of errors?" The result was a graph of the average project error rate versus that of the current project. The system could analyze the error data and predict key project information. If the error rate was abnormally low, such an analysis might display three possible causes: insufficient testing, experienced team, or problem less difficult than expected. The results of prediction would be a graph showing the extrapolated error rate at project conclusion. Valett also displayed screens from the SME

as it has been currently developed, in which actual versus expected growth in a typical system were contrasted. Valett noted that the system is designed to incorporate dynamic development of models and rules and an improved knowledge of the environment.

In response to a question during the panel session pertaining to the cost of data collection, McGarry said that the overhead cost of collecting SEL metrics was 3 to 4 percent, 8 to 10 percent for processing the data, and up to 25 percent for analysis. No information on Ada maintainability was yet available, and no attempt had been made to incorporate Ada in a real-time system. Asked why the design error rate on first-time Ada projects was higher than on FORTRAN projects, McGarry noted that the FORTRAN design process was highly familiar whereas the Ada design process was new. When asked how increased Ada knowledge was distinguished from experience in an application, McGarry answered that it was not, and that the relative importance of application versus language experience was not yet understood. Responding to the question, "Have you looked beyond technical processes to attitude and institutional roadblocks?", Basili observed that there is no current institutional motivation for reuse; in fact, there is motivation (in contracts) for non-reuse.

SESSION 2 - SOFTWARE MODELS

In the first presentation, Robert Tausworthe of the Jet Propulsion Laboratory (JPL) likened the process of software development to a noisy channel (A Communication Channel Model of the Software Process). Over this channel--composed of people and hardware--information flows, is transformed, distorted, erased, delayed, and otherwise modified. The problems with communication channels, Tausworthe said, are high costs, too long a delay between need and satisfaction,

and a need-to-satisfaction correlation that is too difficult to compute. To cope with noisy channels, it is necessary to

- Measure and characterize the channel's parameters
- Expect transmission noise
- Design throughput below channel capacity
- Make information resilient to channel disturbances
- Transmit with greatest signal force possible
- Reduce noise
- Use feedback to correct errors

As axioms, Tausworthe stated that a mapping exists between input requirements and output; that information is not created, it is transformed or lost; that intelligence in the channel contributes to the transformation and noise energy; and that the product yield results from the minimum product specification plus the minimum for all reused parts. In a product-builder channel model, the amount of information into the design engine and factory are the same, but are transformed. The knowledge base and catalog of inputs represent the transform engine; to the extent these are supplied by automation, productivity increases. Using the axioms, Tausworthe derived a formula for production capacity in which the degree of reuse would ultimately determine the bound on productivity. In summary, Tausworthe indicated that the area for productivity improvement was limited, and that the language advantage grows as long as the average yield of reusable parts can be made to increase.

Troy Henson of IBM was the next speaker (Knowledge-Based Assistance in Costing the Space Station Data Management System). He noted the many complex factors that affect a software cost estimate: historical data, software size, productivity, complexity, schedule, project constraints, and criticality. The problem, Henson declared, was to increase the productivity and reliability of software cost estimation

(SCE) by defining the process, automating the methodology, and providing SCE courses. Currently, he said, SCE courses are offered at IBM, and nine algorithmic PC-based tools and a Lotus tool have been developed. The prototype tools include two expert systems: the Software Complexity Determination Assistant and the Software Criticality Assistant.

The Space Station Data Management System (SSDMS) posed special problems in cost estimation due to its long life, remote integration, distributed environment, phased technology insertion, etc. A costing methodology was defined in which requirement specifications were translated to functions. For each function, the size in LOC, criticality, complexity, and release designation were specified. Productivity and verification factors were computed, and the man-months required to accomplish the task were calculated.

Henson concluded that using the SCE tools in costing the SSDMS improved efficiency, accuracy, and consistency. The tools provided a foundation that may be calibrated and expanded to include other areas of software system engineering process control. Responding to questions, Henson noted that their SCE models and database were more relevant than COCOMO for their particular project costs.

The sixth speaker was William Cheadle of Martin Marietta (Software Sizing, Cost Estimation and Scheduling). Cheadle said that Martin Marietta (MMC) has been looking into software development for 15 years and has studied the total life cycle from the definition of a system through final integration. A number of parametric models are in use at MMC: two versions of parametric cost estimation models, a maintenance model, a performance measurement model, a sizing model, a CSCI/CPCI integration model, a risk analysis

simulation tool, and a software architecture sizing and estimating tool. A large database has been accumulated over the 15-year period; it currently contains information on over 53 projects.

The costing profile based on this historical data shows that, on older projects developing "spaghetti" code, 25 percent of the total effort was expended by critical design review (CDR), whereas projects using top-down methodology expended 45 to 55 percent. Analysis of the data also reveals that one SLOC required an average of 2.24 hours of effort when computed over the full project life cycle. Cheadle added that an Ada SLOC is computed at MMC by counting semicolons. In response to a question, Cheadle noted that data from projects using rapid prototyping were going into the models, resulting in significant cost changes.

SESSION 3 - STUDY OF SOFTWARE PRODUCTS

Hasan Sayani from ASTEC (Reverse Engineering: An Aid to Understanding) was the first speaker of the afternoon session. Sayani defined reverse engineering as working backward from any phase in the development life cycle. Without supporting documentation, he said, the process of reverse engineering is somewhat akin to archaeology. Its success depends on recognizing that information may be lost and that ambiguities are inevitable. Reverse engineering may be performed to (1) understand the current system; (2) maintain or change the current system; (3) determine where enhancements to the system are needed and what their effects would be; (4) merge one system with another by defining the common data and interfaces; and (5) inject new technology (e.g., a DBMS).

Sayani described a tool containing an interpreter that accepts source code and generates program specification language (PSL) statements. The program abstraction is then stored in a database from which reports may be generated.

CASE tools may be used to produce a diagram of the system, and other relevant data may be merged.

Through reverse engineering, it is possible to examine the translated language to learn the architecture of procedure calls and data structures. It is also possible to synthesize desired aspects across code units and to pinpoint problems. Sayani noted that, on one very large system under maintenance, an 8-to-1 savings using the reverse engineering tool was observed. Rework was vastly reduced since the ripple effects of modifications could be predicted.

Sayani noted potential pitfalls in the reverse engineering process, such as unexpected code constructs, differences in programming styles, and diverse organizational standards. He predicted that future technology would adapt to broader source code input, produce sophisticated models across languages, have better CASE interfaces, and regenerate code.

The next speaker, Jairus Hine of JPL, presented a case study of Ada projects at JPL (Ada Software Productivity Analysis), where two main databases are used to record the size and cost of software development: a NASA historical database with 10 projects, and a JPL database with 4 projects. In the JPL database, one project caught the eye of the researchers; it had the highest productivity of all the projects examined, and it used Ada. The problem was to determine how much of the effect was due to Ada use. Hine first examined subsystems within this first project, then compared Project 1 with Project 2, a similar system written in Pascal.

Project 1 contained 500,000 LOC and used Ada and C in a prototype environment. Each subsystem used different amounts of Ada. The project was straightforward and was characterized by good communication between users and competent developers. A general rise in productivity was initially

observed as the percent of Ada used in a module increased; however, productivity on the subsystem tasks differed greatly. This Hine attributed to differences in Ada experience and tool and rule availability from one task to the next. Adjusting for these environmental factors, the productivity of the tasks was seen to be very similar, regardless of the amount of Ada used. When Hine grouped the tasks into two categories, primarily Ada and primarily non-Ada, he observed an increase of 2 LOC per day (15 percent overall) in productivity in the Ada group. However, the normalized productivity in Project 1 was found to be considerably lower than that of the Pascal Project (7.4 vs 13.5 SLOC/day). Part of this, Hine added, could be due to the Ada learning curve and other unadjusted environmental factors. Hine hypothesized that, given experienced programmers, a 10- to 25-percent increase in productivity would be possible with Ada. Similar productivity gains, he suggested, were possible with languages other than Ada using modern, modularized design methods.

The final speaker of the session, Robert LaBaugh of Martin Marietta, discussed a project that successfully used Ada in an embedded application for real-time control of a robot arm (Experiences with Ada in an Embedded System). The objectives were to use Ada, evaluating such features as tasking performance, and to develop a generalized control system based on the NASA reference model for control architecture, NASREM. The application concentrated on the two lowest levels of the NASREM architecture: the servo level, which is closest to the hardware, and the primitive level. The system, including all low-level hardware interfaces and controllers, was developed as 11 tasks coded in standard Ada. System performance was more than adequate; all NASREM levels were able to execute within a single 20-millisecond control loop.

La Baugh reported the following as lessons learned from the project: The ability to test/debug on a host does not eliminate testing on the target machine. Portability with Ada is not automatic; there are differences in tasking implementations. Public domain packages need support; the math library worked on the development machine, but machine-specific parameters produced errors on the target machine. Resistance was encountered to using Ada alone for embedded real-time applications, both from "experts" who had heard that Ada was insufficient and from compiler and real-time kernel vendors.

LaBaugh was asked to comment on the statement of experts that tasking cannot be used in embedded systems. He responded that tasking worked, and that using the delay statement to simulate time-slicing fixed the problem encountered when the system was ported to the target machine. Responding to further questions, LaBaugh noted that their design was ad hoc, based on the NASREM architecture, and that, although reuse could be effective in defining NASREM layers, they could not use generic packages while maintaining separate task priorities.

SESSION 4 - TOOLS

The final session was introduced by Mike Gardner of CSC. Gardner noted that the introduction of object-oriented programming and design has raised the question, "Do we continue to use functionally oriented methods in the requirements analysis phase, or should we be moving to object-oriented techniques?" This issue, he said, was the main topic to be addressed in the fourth session.

Daniel Drew began the session with a discussion of the method employed for requirements analysis at Unisys (A Practical Approach to Object Based Requirements Analysis).

As a maintenance organization for the shuttle, Unisys in Houston is interested in using Ada to rewrite or replace existing software. They organize all requirements information into a notebook and use the data to generate a baseline requirements list (BRL). From the BRL, a static entity-relationship (ER) model and object data flow diagrams (ODDs) are created. The object-oriented design is then coded, tested, and delivered.

Drew noted that organizing the BRL forces examination of each requirement. Automated tracking of this list was essential, but simple tools such as a word processor would suffice. Drew said that extracting entities was straightforward, although identifying relationships and attributes and leveling the ER model were not. Drew's group also discovered that naming objects to support the system structure was critical, and that a computer-aided tool was needed to maintain the data dictionary. Drew stated that the problem lies in representing the information, and that database techniques such as ER modeling are appropriate. He added that customer communication needs to be addressed and noted that customers easily understood ODD representations.

Paul Usavage, Jr., of General Electric (GE) was the next presenter (A Modernized PDL Approach for Ada Software Development). The problem, he stated, was to incorporate the benefits of Ada using an automated approach with graphic design tools, while maintaining a high level of risk management. As a result of their investigations, the GE team proposed the following improvements to the software development process:

- Base the software design on the accumulated results of structured analysis
- State requirements with data flow diagrams (DFDs) to aid in understanding the problem

- Design using integrated graphics and program design language (PDL)
- Edit PDL within a graphics context
- Incorporate compiled Ada interfaces
- Perform iterative refinement against graphics and PDL together
- Produce preliminary and as-built design documents automatically
- Use a graphics index to PDL
- Maintain the design database via tools

The team then examined three projects to determine how well these proposals work. Analysis of the first project showed compilation of Ada PDL and control blocks to be inconvenient. Errors uncovered in compilation were mostly in syntax rather than design, and alternative designs became less feasible to generate. Analysis of the second project showed that high-level partitioning based on DFDs worked well and that implementation-level partitioning using PDL and compiled package specifications suffered from rework due to a longer cycle time. The third project used a methodology that was close to that proposed by the team. Analysis showed that the project's object-oriented approach was successful and that the use of graphics worked well.

In conclusion, Usavage noted that graphics and structure charts work better at the high levels of abstraction and that PDL is clearly better at a lower level. He also recommended that a PDL processor be integrated with graphics in a CASE environment. In response to the comment that most people treat structured analysis and object-oriented design as mutually exclusive, Usavage observed that, although it was not easy to go from one to the other, doing so was a powerful tool for understanding the problem.

The last speaker was Jon Franklin Buser of Software Development Concepts (SDC) (Representing Object-Oriented Specifications and Designs With Extended Data Flow Notation). A current goal of SDC, said Buser, is to develop ways to represent object-oriented design and specifications with DFDs. DFDs have certain advantages: they are supported by many CASE tools, they are neither language nor operating-system specific, and many software engineers already have a working understanding of the methodology. There are also some problems: CASE tools enforce unique names, which conflicts with component reuse; level-balancing conflicts with building generic components that have unused access functions; and commonly used partitioning strategies can lead to the loss of the concept of software objects.

Using the example of a simple data storage and reporting system, Buser suggested several new partitioning conventions for representing objects:

- Group all processes that operate on the same real-world object
- Group all data flows associated with the same process or routine
- Name the combined flow for access routines
- Use a double arrow for access routine I/Os

Buser showed an improved diagram for the sample system, in which two-way flows were named to identify the object with which the flow was associated. He concluded by stating that more work with these conventions was needed. CASE tools should be enhanced to support reuse and inheritance, whereas they currently defeat these efforts. Browsers are needed for scanning libraries of reusable components documented by DFDs. Asked if it is difficult to get people to think in terms of objects, Buser responded that by following the

development methodology that SDC taught--first building an information model, then examining the behavior patterns of the objects using state-machine diagrams, and lastly building the process models--it was possible to sidestep issues of an established mindset.

In the panel session, the question was raised as to how to group objects with functions correctly. Every store and flow is a candidate operation for an object, Usavage contributed, noting that a colleague has developed a mechanical transfer process changing arrows to bubbles and vice versa. Mike Gardner then asked if the Unisys approach was not removing information by not showing operations in some way, with which Drew agreed, although he felt that functionality was apparent in an ODD.

PANEL #1

STUDIES AND EXPERIMENTS IN THE SEL

F. McGarry, NASA/GSFC
V. Basili, University of Maryland
J. Valett, NASA/GSFC

EVOLVING IMPACT OF ADA ON A PRODUCTION SOFTWARE ENVIRONMENT

F. McGarry (NASA/GSFC)
L. Esker (CSC)
K. Quimby (CSC)

1.0 BACKGROUND (Chart 1)

Since 1985, the Software Engineering Laboratory (SEL) has been studying the impact of Ada and Ada-related technologies on the software development of production projects within the Flight Dynamics Division (FDD) at NASA/GSFC. Until then, all software development projects had used FORTRAN as the primary implementation language. The Ada development work began with a pilot project and a research project that paralleled a production FORTRAN development project (References 1 and 2). After this initial Ada experience, several later production projects were developed in Ada. For each project, the SEL has collected such detailed information as resource data, error data, component information, methodology, and project characteristics, so that the SEL could study the evolution of the use of Ada itself and the actual characteristics of the Ada development process (Reference 3).

Analysis of the Ada projects has led personnel to document lessons learned during the development of Ada projects (References 4 through 7). These lessons have provided valuable insight into the impact of Ada, especially in the following areas:

1. The impact of Ada on the software development process, that is, the impact Ada has on such measures as productivity, reliability, and maintainability.

2. The impact of Ada over time, as shown by the differences between the first, second, and third Ada projects.

3. The use of Ada and Ada features as the development environment gains more experience in using Ada.

4. The timeframe for realizing the benefits of using Ada.

1.1 ADA PROJECTS STUDIED (Chart 2)

Ada use within the FDD began in January 1985 with the GRODY project. As part of the preparation for developing this system, personnel first participated in a practice Ada project by implementing an electronic mail system (EMS). These two projects actually represent a first Ada experience.

After the GRODY project was well under way, two new Ada simulator projects for the GOES satellite began. GOADA, the dynamics simulator, and GOESIM, the telemetry simulator, collectively represent a second major experience with Ada. They are considered second projects because (1) some team members had previous experience in developing systems in Ada and (2) these two projects could draw on lessons learned from GRODY. Not only were the staffing profiles of the two GOES simulator teams different from the GRODY team, but the two GOES teams began using additional software tools available within the DEC Ada development environment.

Late in 1987 and 1988, two more projects, UARSTELS and Build 4 of FDAS began; these projects represent a third distinct Ada experience. Currently, two more Ada projects are in their early stages: EUVEDSIM and EUVETELS, but these projects are very early in their lifecycles and are not yet available for study.

1.2 PROJECT STATUS AND CHARACTERISTICS (Chart 3)

All totaled, Ada has been used on eight projects in the flight dynamics area. Two projects (EMS and GRODY) are completed; three (GOADA, GOESIM, and FDAS) are well into system testing; and one (UARSTELS) is in the implementation phase. The other two projects (EUVEDSIM and EUVETELS) are in the early requirements analysis phase. These projects range from nearly 6K to 163K SLOC in size, where SLOC is total source lines of code including comments, blanks, newly developed code, and reused code. These projects have required or are expected to require from 4 to 36 months to complete and had from three to seven people working on them. Although GRODY lasted for 36 months, it should be noted that most personnel on this project did not work fulltime on its development. The small EMS project could have been completed by 2 or 3 people; but since it was part of the Ada training for the GRODY project, all GRODY developers participated in some part of the EMS project.

2.0 ADA EVOLUTION

2.1 TEAM EXPERIENCE AND DEVELOPMENT ENVIRONMENT (Chart 4)

Of the eight Ada projects currently under way, six projects have progressed far enough to be studied: EMS, GRODY, GOADA, GOESIM, FDAS, and UARSTELS. All six of the projects studied have been staffed with personnel with a similar level of software development experience, an average of 4 to 5 years. Except for UARSTELS, each project also had personnel with a similar level of experience in the application. To date, the SEL has not observed any impact due to differences in team experience between projects.

It is also too early to observe any differences in the effect of varied levels of Ada experience on project development. The number of people who are formally trained in Ada and/or the number of those who have been on previous Ada projects is still too small. Only the first Ada projects have been completed. Some personnel on those projects have contributed to current, ongoing projects; however, there are not enough people in the environment, even on the most recent Ada projects, to significantly change the ratio of experienced Ada personnel to those with no Ada experience.

The use of tools has evolved somewhat from the first Ada projects. The practice Ada project (EMS) had only rudimentary tools available (compiler, linker, editor). GRODY made use of the DEC symbolic debugger (SD), and the Configuration Management System (CMS). All subsequent Ada projects are using these tools as well as the Language Sensitive Editor (LSE). Project personnel have also developed some additional tools in house to create package bodies and templates for the associated subunits they need to develop.

2.2 SOFTWARE CHARACTERISTICS (Chart 5)

Traditionally, software size has been described in terms of the lines of code developed for the system. However, software size can be expressed by many other measurements (Reference 8), including

1. Total physical lines of code (carriage returns)

2. Noncomment/nonblank physical lines of code

3. Executable lines of code (ELOC) (not including type declarations)

4. Statements (semicolons in Ada, which include type declarations)

Chart 5 describes the size of the Ada projects in the flight dynamics area using these four measurements. The FORTRAN project, GROSS, was also included in the summary for comparison. The GROSS project is the FORTRAN implementation of the GRODY project, and the GRODY/GROSS comparison has been detailed in previous papers. Because the GOESIM and UARSTELS projects are both telemetry simulators, they are also very similar in terms of their functionality. These two Ada projects are estimated to be between 75 and 78 thousand lines of code (KSLOC). In comparison, a typical telemetry simulator in FORTRAN consists of approximately 28 KSLOC.

Unless one counts only Ada statements, these figures tell us that the use of Ada results in many more lines of code than the use of FORTRAN. The increase in lines of code is not necessarily a

negative result. Rather, it is simply that the size of the system implemented in Ada will be larger than an equivalent system in FORTRAN. It is also clear that a precise definition is needed of what is a line of code in Ada and what code is included in that measurement.

Throughout the years of developing similar systems in FORTRAN in the flight dynamics area, the average level of software reuse has been between 15 and 20 percent (Reference 9). FORTRAN projects that attained a 35 percent or higher level of reuse of previously developed code are rare. After the first Ada project and with only 5 to 6 years of maturing in the environment, Ada projects have now achieved a software reuse rate of over 30 percent. This is already greater than the typical FORTRAN project. The UARSTELS project is expected to consist of more than 40 percent reused code. This trend of increasing software reuse is very promising.

2.3 LIFE-CYCLE EFFORT DISTRIBUTION (Chart 6)

The GROSS project followed the typical FORTRAN life-cycle effort distribution (Reference 10). Specifically, a small amount (8 percent) of the total effort expended on the project was spent during the pre-design or requirements analysis phase of the project; 27 percent of the effort was spent during the design phase, 40 percent during the code implementation phase; and 25 percent during the system testing phase. For the Ada projects, significant changes to the life cycle have not yet been observed. However, the Ada life cycle is changing slightly with each project and may soon show a different life cycle than that expected for a FORTRAN project. The life cycles for the second and third Ada projects are shifting slightly to show more design

time required with less system test time.

As the Ada environment matures and the SEL learns more about Ada, the life cycle is expected to continue shifting in the direction that the early literature has reported (Reference 11): more time spent in the design phase and less time in the system test phase. FORTRAN projects could assume the reuse of the life cycle based on past experience. This life cycle cannot be automatically reused in Ada, and more study is needed to determine the duration and products of each phase of an Ada project.

With the current projects, the SEL has not observed significant changes to the life-cycle phases. However, effort by phase is time driven. The SEL also collects effort data by activity across all phases. With this data the amount of effort spent on such activities as design, coding, and testing is very different than the distribution of effort on activities for FORTRAN projects. Much more time is spent on design for the Ada projects, but more analysis is still needed in this area.

2.4 ADA COST/PRODUCTIVITY (Chart 7)

Discussions on Ada productivity are somewhat confusing because so many interpretations exist of software size measures in Ada. Depending on the measurement used and an individual's inclination, one could determine that Ada is either as good or not as good as FORTRAN. Using the total lines of delivered code as a measure, the first, second, and third Ada projects show an improving productivity over time, and they show a productivity greater than FORTRAN. However, considering only code statements (excluding all comments and continued lines of code), the results are different. An increasing productivity trend remains in the

Ada projects over time, but the Ada projects have not yet achieved the productivity level of FORTRAN projects.

Within the flight dynamics environment, many software components are reused on FORTRAN projects. Since no Ada components existed previously, the first Ada projects were, in fact, developing a greater percentage of their delivered code than the typical FORTRAN project. Based on a past study by the SEL and on experience with FORTRAN projects, personnel concluded that reused code costs around 20 percent of the cost of new code (ref 15). The cost of reused code lies in the effort needed to test, integrate, and document the reused code in the new system. Using this estimate, reusability can be factored into software size by estimating the amount of developed code. Because of the differences in cost of new and reused code, developed code is calculated as the amount of new code plus 20 percent of the reused code. With software reusability factored in, the productivity for developed statements on Ada projects is approximately the same as that for FORTRAN projects.

The trends in Ada productivity are very positive. Again, lines of code must be clearly defined when discussing productivity. Using total number of lines as the measurement of software size, Ada productivity was always greater than FORTRAN productivity. However, due to the greater number of lines of an Ada project compared to a similar FORTRAN project, this measure can be misleading.

2.5 USE OF ADA FEATURES (Chart 8)

It is difficult to tell whether a given project really used the Ada language to its fullest capacity. Different applications may or may not need all the features available in Ada. However, in an effort to achieve some measurement in the use of the features available in the Ada language, the SEL identified six Ada features to monitor: generic packages, type declarations, packages, tasks, compilable PDL, and exception handling. The SEL then examined the code to see how little or how much these features were used.

The numbers of packages and type declarations were normalized to the size of the system, and the number of generic packages was divided by the total number of packages in the system. As seen in chart 8, the use of four of these features has evolved over time: generic packages, type declarations, packages, and tasking. Compilable PDL and exception handling did not show any trends. Perhaps it is too early to see results in these areas.

The average size of packages (in SLOC) for the first Ada projects is much higher than the average size of packages for the second and third Ada projects. This is due to a difference in the structuring method between the first Ada projects and all subsequent Ada projects (Reference 4). The first Ada projects were designed with one package at the root of each subsystem, which led to a heavily nested structure. In addition, nesting of package specifications with package bodies was used to control package visibility. Current Ada projects are utilizing the view of subsystems described by Grady Booch (Reference 12) as an abstract design entity whose interface is defined by a number of separately compilable packages, and nesting of Ada packages is limited to generic package instantiations.

The use of generic packages from the first to the current Ada projects seems to be increasing. More tham a third of the packages on current projects are generic packages. This higher use of generics reflects both a stronger emphasis on the development of verbatim reusable components and increased understanding of how to effectively utilize generic Ada packages within the flight dynamics area.

The use of strong typing within these software systems is also increasing, as measured by the number of type declarations per KSLOC. With experience, developers are more comfortable with the strong typing features of Ada and are using its capabilities to a fuller extent.

The use of tasking shows the most dramatic evolution over time for any particular Ada feature in the flight dynamics environment; its use has decreased markedly. The first Ada project, GRODY, contained eight tasks. However, from lessons learned on the GRODY project, personnel on subsequent Ada dynamics simulator projects have reduced that number to four tasks. Current telemetry simulator projects require no tasks at all. In the area of tasking, experience has shown that extensive use of this Ada feature is not appropriate for many applications. Although more extensive use of tasking might be very appropriate for other applications, the use of this Ada feature has definitely changed as project personnel have learned to use tasking only in those situations that are appropriate.

2.6 RELIABILITY, ERROR/CHANGE RATE AND CHARACTERISTICS (Charts 9 and 10)

The SEL measures software reliability by the number of changes or error corrections made to the software. For Ada projects, software error and change rates show a very positive trend. While it is too early to observe a definite difference from the FORTRAN rates, the reliability of the Ada projects is at least as good as that of FORTRAN projects. The error and change rates on the Ada projects are also declining over time, a promising trend. The types of errors also show an evolution from first through third Ada projects.

On a typical FORTRAN project, design errors amount to only 3 percent of the total errors on the project. For the first and second Ada projects, 25 to 35 percent of all errors were classified as design errors, a substantial increase. However, for the third Ada project, design errors are dropping significantly and are estimated to be approximately 7 percent. This rate is close to what is experienced on FORTRAN projects and clearly shows a maturation process with growing expertise in Ada.

Much of the literature on Ada reports that the use of Ada should help reduce the number of interface errors in the software (Reference 13). In our FORTRAN environment, about one-third of all errors on a project are interface errors. On our first and second Ada projects, the number of interface errors was not greatly reduced. Around one-fourth of the errors were interface errors. However, with current projects, the SEL is now seeing the expected results: interface errors are decreasing.

"Errors due to a previous change" is a category of errors that was caused by a previous modification to the software. The first Ada project showed a large jump in the number of these errors compared to those using FORTRAN. However, all subsequent Ada

projects show a rate for "errors due to a previous change" very similar to the FORTRAN rate. Many things probably contributed to this initial jump in the error rate: inexperience with Ada, inexperience with Ada design methodologies, and a nested software architecture that made the software much more complex. Again, the error profile is decreasing with the maturity of the Ada environment.

3.0 OVERALL OBSERVATIONS ON THE IMPACT OF ADA (Chart 11)

In summary, many aspects of software development with Ada have evolved as our Ada development environment has matured and our personnel have become more experienced in the use of Ada. The SEL has seen differences in the areas of cost, reliability, reuse, size, and use of Ada features.

A first Ada project can be expected to cost about 30 percent more than an equivalent FORTRAN project (Reference 14). However, the SEL has observed significant improvements over time as a development environment progresses to second and third uses of Ada.

The reliability of Ada projects is initially similar to what is expected in a mature FORTRAN environment. However, with time, one can expect to gain improvements as experience with the language increases.

Reuse is one of the most promising aspects of Ada. The proportion of reusable Ada software on our Ada projects exceeds the proportion of reusable FORTRAN software on our FORTRAN projects. This result was noted fairly early in our Ada projects, and our experience shows an increasing trend over time.

The size of an Ada system will be larger than a similar system in FORTRAN when considering SLOC. Size measurements can be misleading because different measurements reveal different results. Ratios of Ada to FORTRAN range from 3 to 1 for total physical lines to 1 to 1 for statements.

The use of Ada features definitely evolves with experience. As more experience is gained, some Ada features may be found to be inappropriate for specific applications. However, the lessons learned on an earlier project play an invaluable part in the success of later projects.

REFERENCES

1. Software Engineering Laboratory (SEL), SEL-85-002, Ada Training Evaluation and Recommendations, R. Murphy and M. Stark, October 1985

2. F. McGarry and R. Nelson, "An Experiment with Ada--The GRO Dynamics Simulator," NASA/GSFC, April 1985

3. SEL, SEL-81-104, The Software Engineering Laboratory, D. Card, F. McGarry, G. Page, et al., February 1982

4. --, SEL-88-003, Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis, K. Quimby and L. Esker, 1988

5. --, SEL-88-001, System Testing of a Production Ada Project: The GRODY Study, J. Seigle, L. Esker, and Y. Shi, November 1988

6. C. Brophy, S. Godfrey, et al., "Lessons Learned in the Implementation Phase of a Large Ada Project," Proceedings of the Sixth National Conference on Ada Technology, 1988.

7. SEL, SEL-87-004, Assessing the Ada Design Process and Its Implications: A Case Study, S. Godfrey and C. Brophy, July 1987

8. D. Firesmith, "Mixing Applies and Oranges: Or What Is an Ada Line of Code Anyway?," Ada Letters, September/October 1988

9. Computer Sciences Corporation (CSC), IM-88/083(59 253), Software Reuse Profile Study of Recent FORTRAN Projects in the Flight Dynamics Area, L. Esker, January 1989

10. SEL, SEL-81-205, Recommended Approach to Software Development, F. McGarry, G. Page, et al., April 1983

11. V. Castor and D. Preston, "Programmers Produce More With Ada," Defence Electronics, June 1987

12. G. Booch, Software Engineering With Ada. Menlo Park, CA: Benjamin/Cummings Publishing Company, 1983

13. The MITRE Corporation, Use of Ada for FAA's Advanced Automation System (AAS), V. Basili et al., April 1987

14. B. Boehm, "Improving Software Productivity," Computer, September 1987

15. SEL, SEL-84-001, Manager's Handbook For Software Development, W. Agresti, F. McGarry, et al., April 1984

THE VIEWGRAPH MATERIALS

FOR THE

F. MCGARRY PRESENTATION FOLLOW

EVOLVING IMPACTS OF Ada ON A PRODUCTION SOFTWARE ENVIRONMENT

FRANK McGARRY

NASA/GSFC

LINDA ESKER
KELVIN QUIMBY

COMPUTER SCIENCES CORPORATION

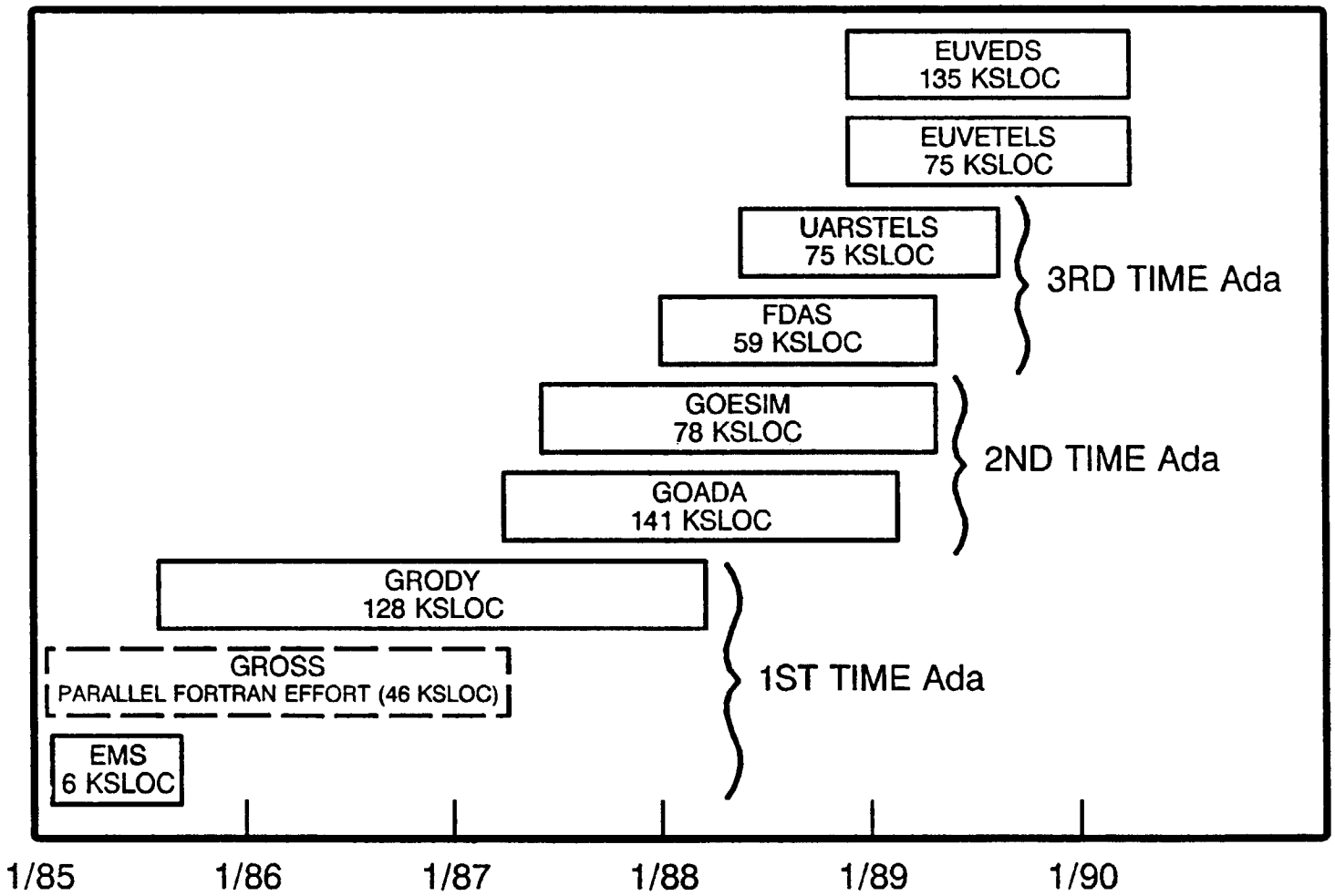
BACKGROUND

- Ada HAS BEEN UTILIZED ON 8 PROJECTS IN FLIGHT DYNAMICS DIVISION AT NASA/GSFC
- DETAILED DATA HAS BEEN COLLECTED/STUDIED FOR ALL PROJECTS - BY THE SEL
- APPROACH TO TRAINING/DESIGN/IMPLEMENTATION/TESTING HAS EVLOVED

POINTS TO BE ADDRESSED

- WHAT IS THE IMPACT OF Ada ON DEVELOPMENT PROFILES (E.G. EFFORT DISTRIBUTION)?
- WHAT ARE EFFECTS IN PRODUCTIVITY/RELIABILITY/MAINTAINABILITY...?
- WHAT IS THE CHANGING IMPACT OF Ada - 1ST TIME, 2ND TIME, 3RD TIME?
- DO WE USE Ada "DIFFERENTLY" OVER TIME...(BETTER %)?
- HOW LONG DOES IT TAKE TO REAP PROMISED BENEFITS?

Ada PROJECTS IN FLIGHT DYNAMICS DIVISION



F. McGarry
NASA/GSFC

D217.003

Ada PROJECTS STUDIED

PROJECT	TYPE	SIZE* (SLOC)	START DATE	DURATION	(11/30/88) STATUS	STAFF LEVEL
EMS	ELECTRONIC MAIL (PRACTICE/TRAINING)	5730	3/85	4 MO.	COMPLETE	7
GRODY	SIMULATOR (FLIGHT CONTROL SYSTEM)	128000	8/85	36 MO.	COMPLETE	7
GOADA	SIMULATOR (FLIGHT CONTROL SYSTEM)	141000	7/87	20 MO.	SYSTEM TEST	7
GOESIM	SIMULATOR (TELEMETRY)	78000	9/87	18 MO.	SYSTEM TEST	4
FDAS	EXECUTIVE (SOURCE CONTROL)	58700	1/88	13 MO.	SYSTEM TEST	4
UARSTELS	SIMULATOR (TELEMETRY)	75000	2/88	18 MO.	CODE	3

*SLOC = TOTAL LINES (CARRIAGE RETURNS) INCLUDES COMMENTS/BLANKS/REUSED
ALL PROJECTS DEVELOPED ON DEC VAX 11/780 OR VAX 8600

D217.0104

TEAM EXPERIENCE AND ENVIRONMENT USED

PROJECT	APPLICATION EXPERIENCE (RATIO)	AVERAGE S/W EXPERIENCE (YEARS)	Ada EXPERIENCE (RATIO)		ENVIRONMENT** COMPILABLE PDL	LSE	SD	CMS
EMS	NA	4.7	0		N	N	N	N
GRODY	1/7	4.7	0*		N	N	Y	Y
GOADA	2/7	5.9	3/7		Y	Y	Y	Y
GOESIM	1/4	5.7	1/4		Y	Y	Y	Y
FDAS	0/4	4.4	2/4		N	Y	Y	Y
UARSTELS	3/3	5.5	1/3		Y	Y	Y	Y

*TEAM HAD DEVELOPED SMALL Ada PRACTICE PROBLEM
**ALL DEVELOPMENT USED DEC.

SOFTWARE CHARACTERISTICS

	GROSS (FORTRAN)	GRODY (1ST TIME Ada)	GOADA (2ND TIME Ada)	GOESIM (2ND TIME Ada)	FDAS (3RD TIME Ada)	VARSTELS (3RD TIME Ada)	TYPICAL TM SIMULATION FORTRAN
TOTAL LINES (CR)	45500	128000	139000	78000	58700	75000	28000
NON COMMENT/ NON BLANK	26000	60000	68500	36000	31300	-	15000
EXECUTABLE LINES (NO TYPE DECL)	22500	40250	42000	21000	17100	-	12500
STATEMENTS (SEMI-COLON INCLUDES TYPE DECL)	22300	22500	25000	14000	11000	-	12000
% REUSED	36%	0	38%	32%	NA	42%	15%

1. Ada RESULTS IN LARGER SYSTEM (SLOC)
2. REUSE TREND VERY POSITIVE
3. "LINE OF CODE" DEFINITION CRITICAL

D217.006

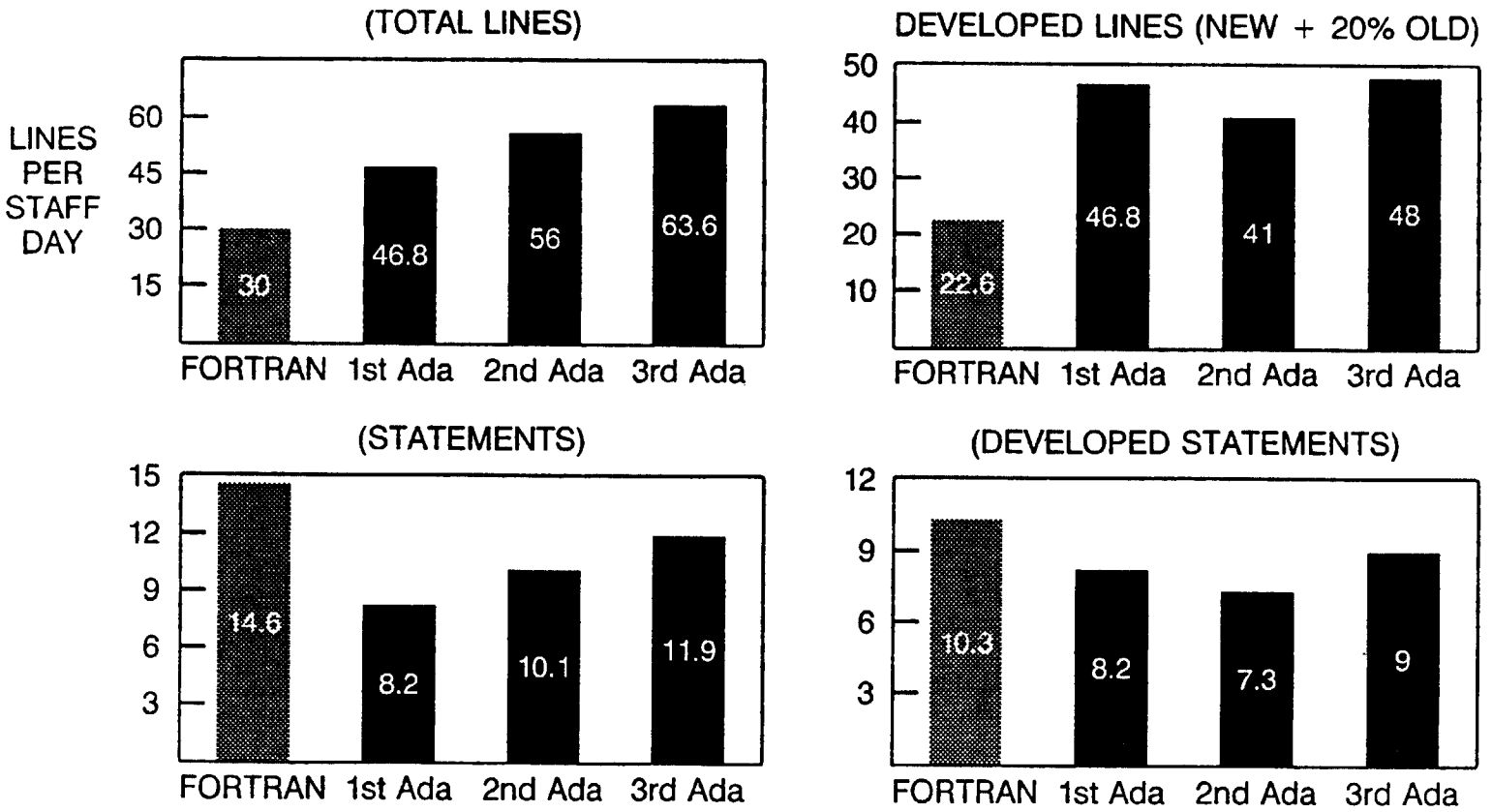
Ada IMPACTS ON LIFE CYCLE EFFORT DISTRIBUTION

	% TOTAL EFFORT*					
	GROSS (FORTRAN)	GRODY (1ST TIME Ada)	GOADA (2ND TIME Ada)	GOESIM (2ND TIME Ada)	FDAS (3RD TIME Ada)	UARSTELS (3RD TIME Ada)
PRE DESIGN	8	8	6	4	8	-
DESIGN	27	24	32	34	34	-
CODE	40	42	42	41	38	-
TEST	25	26	20	21	20	-
TOTAL EFFORT (HOURS)	12150	21860	21230**	10360**	7390**	-

SIGNIFICANT CHANGES TO LIFE CYCLE HAVE NOT YET BEEN OBSERVED - BUT . . .

*EFFORT DISTRIBUTION BASED ON PHASE DATES (E.G. END DESIGN, END CODE, END TEST)
** PARTIALLY BASED ON ESTIMATES TO COMPLETION

Ada COST/PRODUCTIVITY

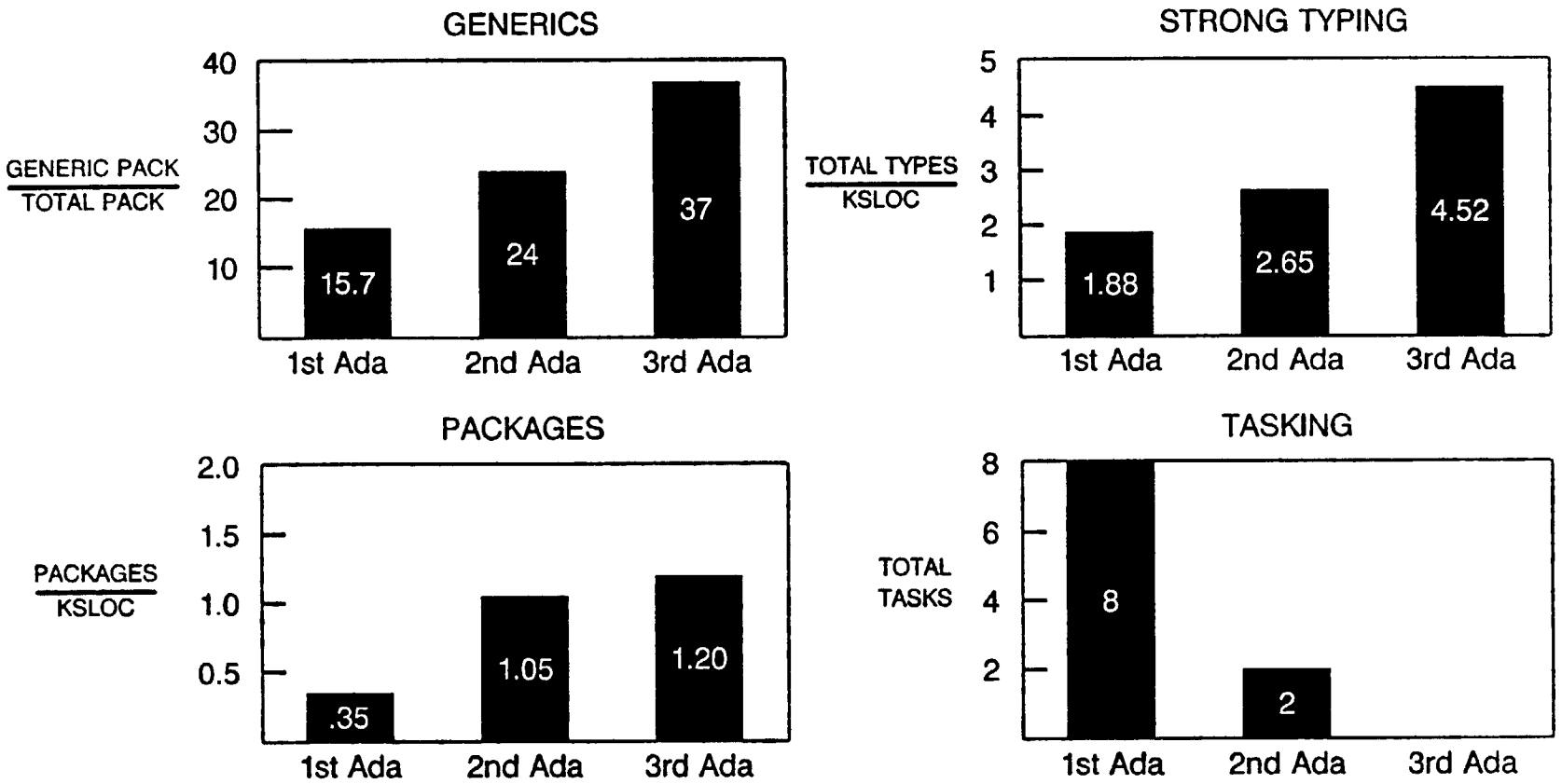


- CLEARLY DEFINE "LINES OF CODE"
- DO NOT USE SLOC IN COMPARING FORTRAN/Ada
- Ada TRENDS ARE IN POSITIVE DIRECTION

(GROSS/GRODY/GOADA/GOESIM/FDAS)

D217.016

USE OF Ada FEATURES



- USE OF Ada FEATURES CHANGES APPRECIATELY WITH EXPERIENCE
- NOT ALL FEATURE APPROPRIATE FOR APPLICATION

F. McGarry
NASA/GSFC

D217.018

Ada AND ERROR/CHANGE RATE

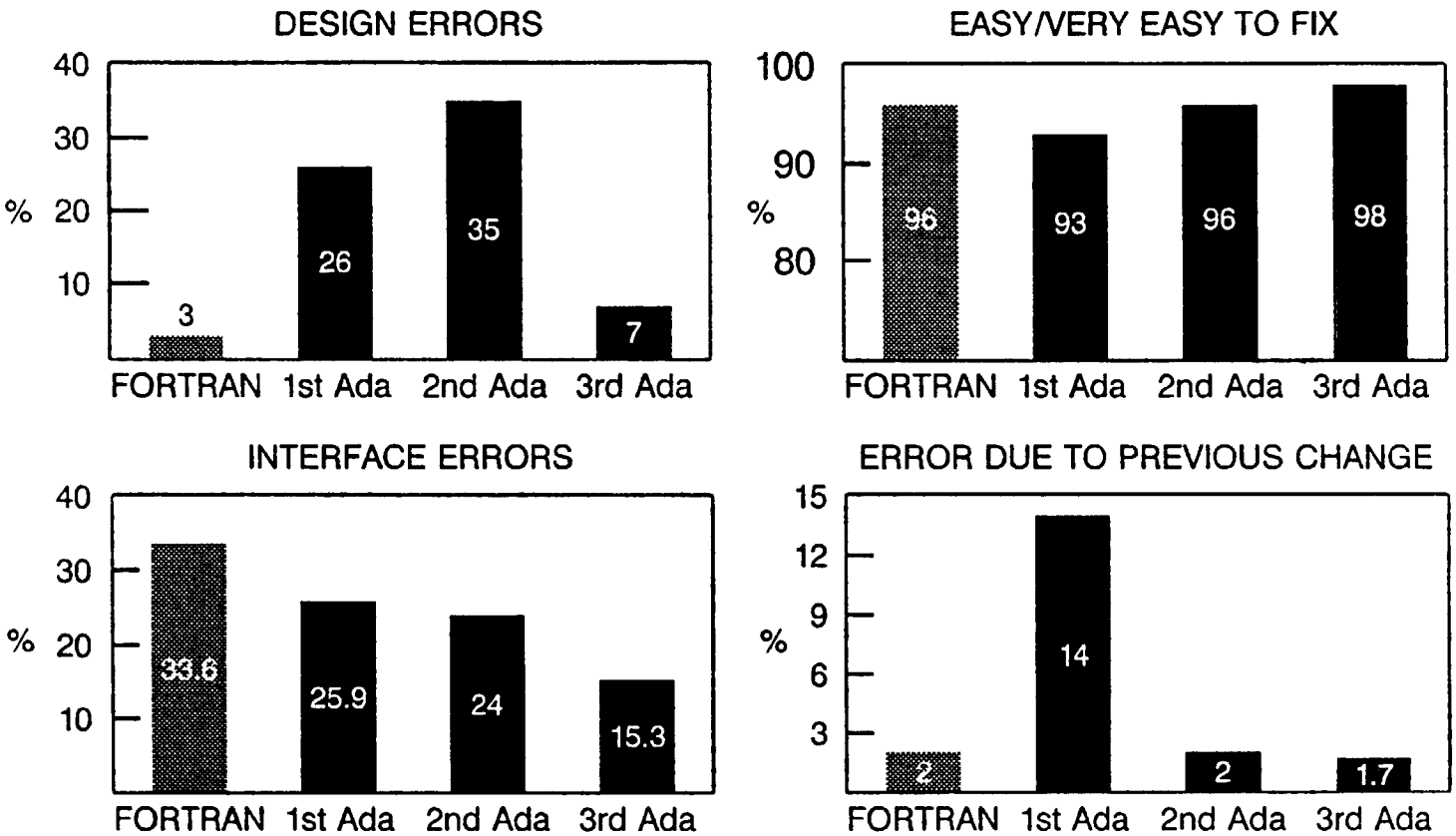
	GROSS (FORTRAN)	GRODY	GOADA	GOESIM	FDAS
CHANGES/KSLOC*	5.8	4.2	2.8	2.4	6.8
ERRORS/KSLOC	3.4	1.8	1.7	1.4	1.0

- RELIABILITY OF Ada SOFTWARE - AT LEAST AS GOOD AS FORTRAN
- VERY POSITIVE TRENDS FOR Ada - OVER TIME

*SLOC = TOTAL LINES (INCLUDES COMMENTS/REUSED)

D217.010

ERROR CHARACTERISTICS



- Ada ERROR PROFILE CHANGES WITH MATURITY OF USE
- Ada HELPS CUT INTERFACE ERRORS

EVOLVING IMPACTS OF Ada

OBSERVATIONS
(FROM 6 PROJECTS IN THE SEL)

1. COST	• 30%+ OVERHEAD TO "FIRST TIME" PROJECTS • SIGNIFICANT IMPROVEMENTS ON 2ND TIME/3RD TIME USE
2. RELIABILITY	• INITIALLY SIMILAR TO FORTRAN • IMPROVEMENTS WITH EXPERIENCE
3. REUSE	• VERY POSITIVE TRENDS - EXCEEDS FORTRAN EXPERIENCE EARLY
4. SIZE	(Ada TO FORTRAN - LARGER) • TOTAL LINES 3 TO 1 • EXECUTABLE LINES 2 TO 1 • NON COMMENTS 2 1/2 TO 1 • STATEMENTS 1 TO 1
5. USE OF Ada FEATURE	• PROMINANT EVOLUTION WITH "EXPERIENCE" • SEEMS RELATED TO IMPROVED DEVELOPMENT • SOME FEATURES INAPPROPRIATE TO SPECIFIC PROBLEMS

USE OF ADA FEATURES

	EMS*	GRODY	GOADA	GOESIM	FDAS*	UARSTELS
COMPILABLE PDL	N	N	Y	Y	N	Y
GENERICS [GENERIC PACKAGES / TOTAL PACKAGES]	0	15.7%	22%	25.8%	9%	37%
STRONG TYPING [TOTAL TYPES DECLARED / KSLOC]	3.86	1.88	2.37	2.96	1.41	4.52
PACKAGES [PACKAGES / KSLOC]	1.05	.35	.95	1.14	1.00	1.20
TASKING	0	8	3	0	1	0
EXCEPTION HANDLING [EXCEPTIONS / UNITS]	.41	.15	.24	.17	.10	.15

- USE OF ADA FEATURES CHANGES APPRECIABLY W/"EXPERIENCE"
- NOT ALL FEATURES APPROPRIATE FOR SPECIFIC APPLICATION
- LESSONS LEARNED' ANALYSIS EFFECTIVE (eg. TASKING)

*MUCH DIFFERENT APPLICATION THAN OTHERS

ERROR CHARACTERISTICS

	GROSS (FORTRAN)	(1ST TIME Ada)	(2ND TIME Ada)	(3RD TIME Ada)
% DESIGN ERRORS	3	25.9	35	7
% "EASY TO FIX + VERY EASY TO FIX"	96	93	96	98
% INTERFACE ERRORS	33.6	25.9	24	15.3
% ERRORS DUE TO PREVIOUS CHANGE	2	14	2	1.7

- Ada ERROR PROFILES CHANGING WITH MATURITY OF "ENVIRONMENT"
- Ada HELPS CUT INTERFACE ERRORS
- FORTRAN DESIGN REUSE APPARENT

D217.011

Ada COST/PRODUCTIVITY

PRODUCTIVITY
(LINES PER STAFF-DAY)

	GROSS (FORTRAN)	GRODY	GOADA	GOESIM	FDAS
TOTAL LINES	30.0	46.8	52.4	60.2	63.6
DEVELOPED LINES (NEW +20% OLD)	22.6	46.8	37.0	45.5	47.8
NON COMMENT	17.1	22.0	25.8	27.8	33.9
STATEMENTS	14.7	8.2	9.4	10.8	11.9
DEVELOPMENT STMTS	10.3	8.2	6.9	8.3	9.0

- CLEARLY DEFINE "LINE OF CODE"
- DO NOT USE SLOC IN COMPARING FORTRAN/Ada
- Ada TRENDS ARE IN POSITIVE DIRECTION

458

N91-10608

UMIACS–TR–88–92
CS–TR–2158

December, 1988

Towards A Comprehensive Framework for Reuse:† A Reuse–Enabling Software Evolution Environment

V. R. Basili and H.D. Rombach
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

MI915766

ABSTRACT

Reuse of products, processes and knowledge will be the key to enable the software industry to achieve the dramatic improvement in productivity and quality required to satisfy the anticipated growing demands. Although experience shows that certain kinds of reuse can be successful, general success has been elusive. A software life–cycle technology which allows broad and extensive reuse could provide the means to achieving the desired order–of–magnitude improvements. This paper motivates and outlines the scope of a comprehensive framework for understanding, planning, evaluating and motivating reuse practices and the necessary research activities. As a first step towards such a framework, a reuse–enabling software evolution environment model is introduced which provides a basis for the effective recording of experience, the generalization and tailoring of experience, the formalization of experience, and the (re-)use of experience.

† Research for this study was supported in part by NASA grant nSG–5123, ONR grant N00014–87–K–0307 and Airmics grant DE–AC05–OR21400 to the University of Maryland.

TABLE OF CONTENTS:

1. INTRODUCTION

The existing gap between the demand and our ability to produce high quality software cost-effectively calls for improved software life-cycle technology. A reuse-enabling software life-cycle technology is expected to contribute significantly to higher quality and productivity. Quality can be expected to improve by reusing proven experience in the form of products, processes and knowledge. Productivity can be expected to increase by using existing experience rather than developing it from scratch whenever needed.

Reusing existing experience is the key to progress in any area. Without reuse everything must be re-learned and re-created; progress in an economical fashion is unlikely. During the evolution* of software, we routinely reuse experience in the form of existing products (e.g. generic Ada components, design documents, mathematical subroutines), processes (e.g., design inspections methods, compiler tools), and domain-specific knowledge (e.g., cost models, lessons learned, measurement data). Most reuse occurs implicitly in an ad-hoc fashion rather than as the result of explicit planning and support. While reuse is less institutionalized in software engineering than in other engineering disciplines, there exist some successful cases of reuse, i.e. product reuse. Reuse in software engineering has been successful whenever the reused experience is self-describing, e.g., mathematical subroutines, or the stability of the context in which the experience is reused compensates for the lack of self-description, e.g., reuse of high-level designs across projects with similar characteristics regarding the application domain, the design methods, and the personnel. In software engineering, the potential productivity pay-off from reuse can be quite high since it is inexpensive to store and reproduce software engineering experience compared to other engineering disciplines.

The goal of research in the area of reuse is the achievement of systematic methods for effectively reusing existing experience to maximize quality and cost benefits. Successful reuse depends on the characteristics of the candidate reuse objects, the characteristics of the reuse process

* The term "evolution" is used in this paper to comprise the entire software life-cycle (development and maintenance).

itself, and the technical and managerial environment in which reuse takes place. Interest in reusability has re-emerged during the last couple of years [4, 9, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21], due in part to the stimulus provided by Ada and in part to our increased understanding of the relation between software processes and products.

Our increased understanding tells us that in order to improve quality and productivity via reuse we need a framework which allows (a) the reuse of all kinds of software engineering experience, i.e., products, processes and knowledge, (b) the better understanding of the reuse process itself, and (c) the better understanding of the technical and managerial evolution environment in which reuse is expected to be enabled.

This paper presents a reuse-enabling software evolution environment model, the first step towards a comprehensive framework for understanding, planning, evaluating and motivating reuse practices and the necessary research activities. Section 2 motivates the necessary scope of a comprehensive reuse framework and the important role of a reuse-enabling software evolution environment model within such a framework. Section 3 introduces the reuse-enabling software evolution environment model and discusses its ability to explicitly model the recording of experience, the generalization and tailoring of experience, the formalization of experience, and the (re-) use of experience. The TAME model, a specific instantiation of the reuse-enabling software evolution environment model, is presented in Section 4. This specific instantiation is used to more specifically describe the integration of the recording and (re-)use activities into an improvement oriented software evolution process.

Before we proceed, we define some crucial terms that will be used in this paper so the reader understands what we mean by them in the software context. We have tailored Webster's general definitions of these terms to the specific domain of software evolution. *Improvement* means enhancing a software process or product with respect to quality and productivity. *Learning* is the activity of acquiring experience by instruction (e.g., construction) or study (e.g., analysis). *Reuse* is the activity of repeatedly using existing experience, after reclaiming it, with or without

modification. *Feedback* means returning to the entry point of some process armed with the experience created during prior executions of the process. We use the expression *experience base* to mean a repository containing all kinds of experience. An experience base can be implemented in a variety of ways depending on the type of experience stored. An experience base may consist of one or more of the following: traditional databases containing factual pieces of information, information bases containing structured information, and knowledge bases including mechanisms for deducing new information [5, 24].

2. SCOPE OF A COMPREHENSIVE REUSE FRAMEWORK

Reuse in most environments is implicit and ad-hoc. When it is explicit or planned, it predominantly deals with the reuse of code. In Section 1, we expressed our belief that effective reuse technology needs to be based on (a) the reuse of products, processes and knowledge, (b) a good understanding of the reuse process itself, and (c) a good understanding of the reuse-enabling software evolution environment.

To better justify these beliefs, we will describe and discuss the reuse practice in the Software Engineering Laboratory (SEL) at NASA Goddard Space Flight Center [2, 18]. This is an example where reuse has been quite successful at a variety of levels, albeit predominantly implicit. Ground support software for satellites has been developed for a number of years in FORTRAN. Reused experience exists in the people, methods, and tools as well as in the program library and measurement database.

To explain reuse in this environment we must first explain the management structure. There are two levels of management involved in the technical project management. The second level managers (one from NASA and one from Computer Sciences Corporation, the contractor), have been managing this class of projects for several years. Specific project managers are typically promoted from within the ranks, on either side, from the better developers on prior projects.

This provides a continual learning experience for the management team. Technical review and discussion is informal but commonplace. Lessons learned from experience are used to improve management's ability to monitor and control project developments.

The organizational structure has been relatively constant from project to project. There have been minor variations due to improvements in such things as methods and tools which have evolved from experience or been motivated the literature and verified by experimental data analysis on prior projects.

The basic systems have been relatively constant. This permits reuse of the application knowledge as well as the requirements, and design. For example the requirements documents are quite mixed with regard to the level of specificity. In some places they are quite precise but in other cases the are very incomplete, relying on the experience of the people from prior projects.

Requirements documents have phrases similar to the following: Capability X for new satellite S2 is similar to capability X for satellite S1 except for the following... This implicitly provides reuse of prior requirements documents as well as implicitly allows for reuse of prior design documents and code.

Systems within a class, all have a similar design at the top level and the interfaces among subsystems are relatively well defined and tend to be relatively error free. Design is implicitly reused from system to system as specified by the experienced high level managers.

Reuse at the code level is more explicit. The software development process used is a reuse oriented version of the waterfall model. The coding phase begins by seeding the code library with the appropriately specified elements from the appropriate prior projects. These code components are then examined for their ability to be reused. Some are used as is, others modified minimally, others modified extensively, and yet others are eliminated and judged easier to develop from scratch. This is a reuse approach that has evolved over time and has been quite effective.

A variety of tools have evolved that are quite application specific. These include everything from tools that generate displays needed for testing to application specific system utilities.

Knowledge about these tools has been disseminated by guidance from more senior members of the development team.

The SEL environment is a good example of strong reuse at a variety of levels, in a variety of ways as part of the software development process. There has been a pattern of learning and reusing knowledge, processes and products. The use of the measurement database has helped with project control and schedule as well as quality assessment and productivity [2, 18].

NASA is now considering changing to Ada. Several Ada projects have already been completed. This has involved an obvious loss in the reuse heritage at the code level, as was anticipated. But it has also involved a less obvious and unexpected loss of reuse at the requirements and design level, in the organizational structure, and even in the application knowledge area.

The initial impact of Ada was staggering because of the implicit, rather than explicit, understanding of reuse in the environment. This understanding of reuse needs to be formalized.

Based upon the concept that reuse is more than just reuse of code and that it needs to be explicitly modeled, we need to reconsider how we measure progress in reuse. The measurements currently used in the SEL are based upon lines of code reused from one project to another. Given this view, progress may not be related at all to the lines of code reused. We need to measure the effects of reuse on the resources expended in the entire software life cycle and on the quality of the products produced using an explicit reuse oriented evolution model. In fact, the process should allow us measure for any set of reuse–related goals [3, 4, 8, 10]. Changing our models and our metrics will help us to better understand the effects of the traditional reuse practices and compare them with the effects of an explicit reuse oriented reuse model.

In summary, we believe that a comprehensive reuse framework needs to include (a) a reuse-enabling software evolution environment model, (b) detailed models of reuse and learning, and (c) characterization schemes for reuse and learning based upon these models.

3. A REUSE-ENABLING ENVIRONMENT MODEL

In the past, reuse has been discussed independent of the software evolution environment. We believe reuse can only be an effective mechanism if it is viewed as an integral part, paired with learning, of a reuse-enabling software evolution environment. None of the traditional engineering disciplines has ever introduced the reuse of building blocks as independent of the respective building process. For example, in civil engineering people have not created "reuse libraries" containing building blocks of all shapes and structures, and then tried to use them to build bridges, town houses, high-rises and cottages. Instead, they devised a standard technology for building certain types of buildings (e.g., town houses) through a long process of understanding and learning. This allowed them to define the needs for certain standard building blocks at well-defined stages of their construction process. In the software arena we have not followed this approach.

If we accept the premise that effective reuse requires a good understanding of the environment in which it is expected to take place, then we must model reuse in the context of a reuse-enabling software evolution environment. Such a context will allow us to learn how to reuse better. The ultimate expectation is that such improvement would lead to an ever increasing usage of generator-technology during software evolution. The ability to automate the generation of products from other products reflects the ultimate degree of understanding the underlying construction processes. Automated processes are easy to reuse. For example, in building compiler front-ends, we rarely reuse components of other compilers; instead, we reuse the compiler generators which automate the entire process of building compiler front-ends from formal language specifications.

In Section 3.1 we discuss how learning and reuse implicitly occur in the context of traditional software evolution environments. In Section 3.2, we discuss how learning and reuse can be explicitly modeled in the context of a reuse-enabling software evolution environment.

3.1. Implicit Learning and Reuse

During a workshop on "Requirements for Software Development Environments", held at the University of Maryland in 1985, a view of a software evolution environment was proposed that consisted of an information system and three information producers and consumers: people, methods, and tools [22]. The information system is defined by a software evolution process model describing the information, the communication among people, methods and tools, and the activity sequences for developing and maintaining software.

The traditional software evolution environment model in Figure 1 is a refinement of this earlier model.

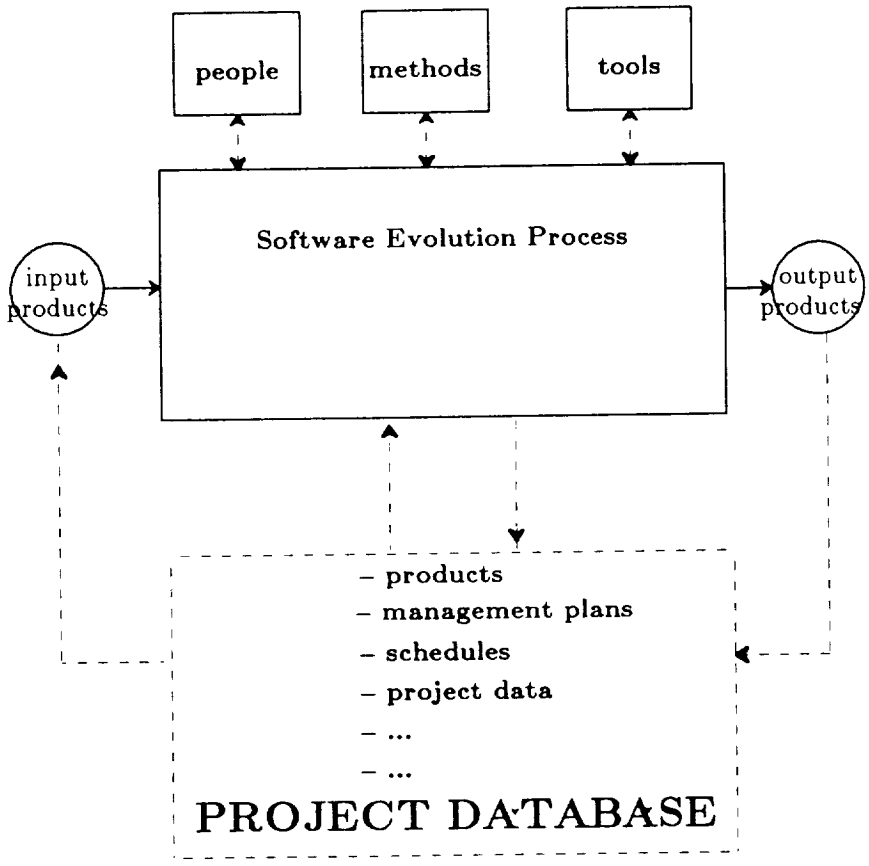


Figure 1: Traditional (non–reuse oriented) Software Evolution Environment Model

The purpose of the software evolution process is to produce output products, e.g., design documents, code, from input products, e.g., requirement documents. People execute this process manually or by utilizing available methods and tools. These methods and tools can be under the control of a project database. All or part of the information produced during this process is stored in a project database, e.g., products, plans such as management plans or schedules, project data.

Typically, support for such a traditional software evolution environment model includes a project database and means for the interaction of people with methods, tools, and the project database during software evolution. The experience of people, as well as some of the methods and tools, is usually not controlled by the project database. As a consequence, this experience is not owned by the organization (via the project database) but rather owned by individual human beings and lost entirely after the project has been completed.

Although the ideas of learning and reuse are not explicitly reflected in the traditional software evolution environment model, they do exist implicitly. The experience of the people involved in the software evolution process and the experience encoded in methods and tools is reused. In many cases, previously developed products are reused as input products. In the same way, products developed during one activity of the evolution process can be reused in subsequent activities of this same process. People learn (gain experience) from performing the activities of the evolution process. Another form of implicit learning occurs whenever products, plans, or project data are stored in the project database.

The basic problem in this traditional environment model is not that learning and reuse can not occur, but that learning and reuse are not explicitly supported and only because of individual efforts or by accident.

3.2. Explicit Modeling of Learning and Reuse

Systematic improvement of software evolution practices requires a reuse-enabling environment model which explicitly models learning, reuse and feedback activities, and integrates them into the software evolution process. Figure 2 depicts such a reuse-enabling environment model.

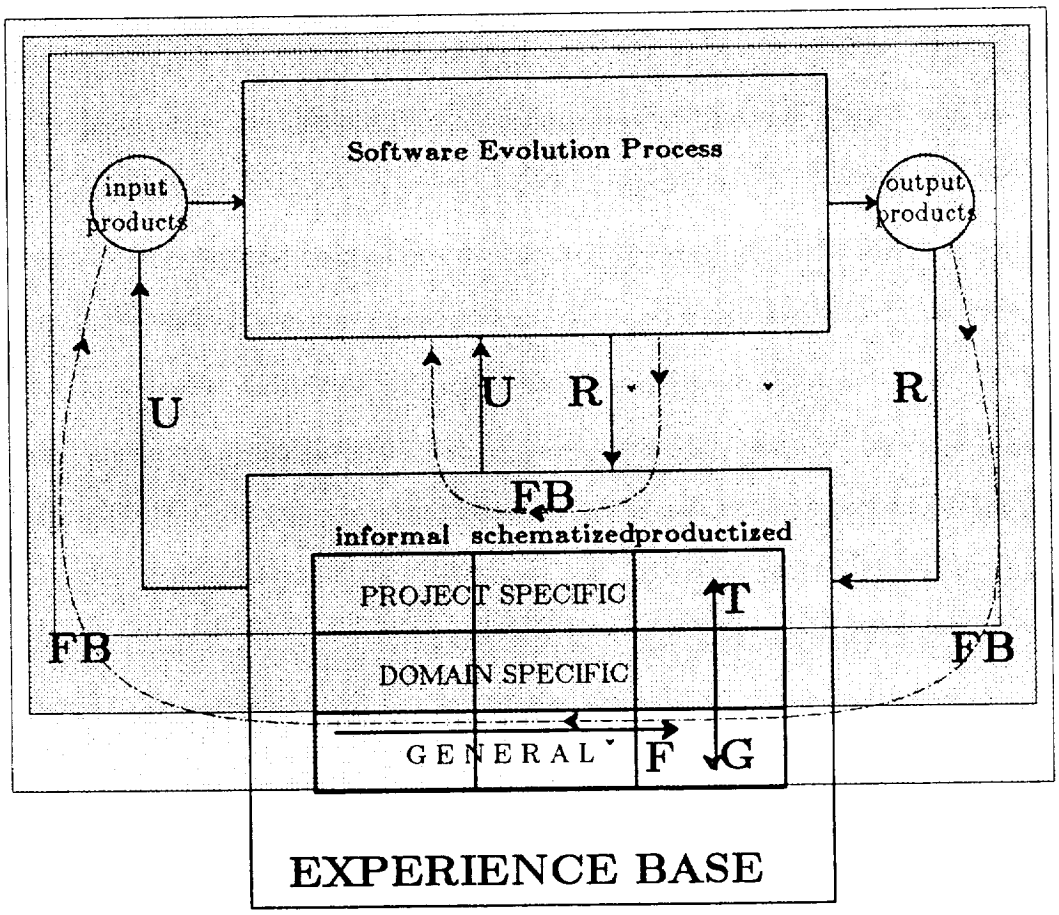


Figure 2: Reuse–Enabling Software Evolution Environment Model

All the potentially reusable experience, including software evolution methods and tools, are under the control of an experience base. Improvement is based on the feedback of existing experience (labeled with "FB" for reuse in Figure 2). Feedback requires learning and reuse. Systematic learning requires support for the recording of experience (labeled with "R" for recording in Figure

2), the off-line* generalizing or tailoring of experience (labeled with "G" and "T" for generalizing and tailoring in Figure 2), and the formalizing of experience (labeled with "F" for formalizing in Figure 2). Off-line generalization is concerned with movement of experience from project-specific to domain-specific and general; off-line tailoring is concerned with movement of experience from general to domain-specific and project-specific. Off-line formalization is concerned with movement of experience from informal to schematized and productized. Systematic reuse requires support for (re-)using existing experience (labeled with "U" for use in Figure 2), and on-line* generalizing or tailoring of candidate experience (not explicitly reflected in Figure 2, because it is assumed to be an integral part of the (re-)use activity).

Although reuse and learning are possible in both the reuse-enabling and the traditional environment models, there are significant differences in the way experience is viewed and how learning and reuse are explicitly integrated and supported. The basic difference between the reuse-enabling model and the traditional model is that learning and reuse become explicitly modeled and are desired characteristics of software evolution.

3.2.1. Recording Experience

The objective of recording experience is to create a repository of well specified and organized experience. This requires a precise description of the experience to be recorded, the design and implementation of a comprehensive experience base, and effective mechanisms for collecting, validating, storing and retrieving experience. We replace the project database of the traditional environment model by an the more comprehensive concept of an experience base which is intended to capture the entire body of experience recorded during the planning and execution of all software projects within an organization. All information flows between the software evolution process and the experience base reflecting the recording of experience are labeled with "R" in Figure 2.

* The attributes "on-line" and "off-line" indicate whether the corresponding activities are performed as part or independent of any particular software evolution project.

Examples of recording experience include such activities as (a) storing of appropriately documented, catalogued and categorized code components from prior systems in a product library, (b) cataloguing of a set of lessons learned in applying a new technology in a knowledge base, or (c) capturing of measurement data related to the cost of developing a system in a measurement database.

In the SEL example of Section 2, code from prior systems is available to the program library of the current project although no code object repository has been developed. Measurement data characterizing a broad number of project aspects such as the project environment, methods and tools used, defects encountered, and resources spent are explicitly stored in the SEL measurement database [2, 8, 18]. Requirements and design documents as well as lessons learned about the technical and managerial implications of various methods and tools are implicitly stored in humans or on paper.

Today it is possible, but not common, to find product libraries. It is even less common to record process-related experience such as process plans or data which characterize the impact of certain methods and tools within an organization. There exist two main reasons why we need to record more process-related experience: (a) it is generally hard to modify existing products efficiently without any knowledge regarding the processes according to which they were created, and (b) the effective reuse of process-related experience such as process plans or data could provide significantly more leverage for improvement than just the reuse of products.

3.2.2. Generalizing & Tailoring Existing Experience Prior to its Potential Reuse

The objective of generalizing existing experience prior to its reuse is to make a candidate reuse object useful in a larger set of potential target applications. The objective of tailoring existing experience prior to its potential reuse is to fine-tune a candidate reuse object to fit a specific task or exhibit special attributes, such as size or performance. These activities require a well-documented cataloged and categorized set of reuse objects, mechanisms that support the

modification process, and an understanding of the potential target applications. Generalization and tailoring are specifically concerned with movement across the boundaries of the "generality" dimension: from general to domain-specific and project-specific and vice versa. Objectives and characteristics are different from project to project, and even more so from environment to environment. We cannot reuse past experience without modifying it to the needs of the current project. The stability of the environment in which reuse takes place, as well as the origination of the experience, determine the amount of tailoring required.

Examples of generalizing and tailoring experience include such activities as (a) developing a generic package from a specific package, (b) instantiating a generic package for a specific type, (c) generalizing lessons learned from a specific design technology for a specific application to any design for that application or any application, (d) or parameterizing a cost model for a specific environment.

In the SEL, requirements and design documents have implicitly evolved to be applicable to all FORTRAN projects in the ground support software domain. Measurement data have been explicitly generalized into domain-specific baselines regarding defects and resource expenditures [2, 8, 18]. Requirements and designs are implicitly tailored towards the needs of a new project based on the manager's experience, and code is explicitly hand-modified to the needs of a new project.

In general, recorded experience is project-specific. In order to reuse this experience in a future project within the same application domain, we have to (a) generalize the recorded project specific experience into domain specific or general experience and (b) then tailor it again to the specific characteristics of the new project. We distinguish between off-line and on-line generalizing and tailoring activities:

- **Off-line generalizing and tailoring** is concerned with increasing the reuse potential of existing process and product-related experience before knowing the precise reuse context (i.e., the project within which the experience is being reused). Off-line generalization and tailoring is

concerned with movement across the boundaries of the specificity dimension within the experience base: from general to domain-specific and then to project-specific, and visa versa. These activities are labeled with "G" and "T" in Figure 2. An example of off-line generalization is the construction of baselines. The idea is to use project-specific measurement data (e.g., fault profiles across development phases) of several projects within some application domain and to create the application-domain specific fault profile baseline. Each new project within the same application domain might reuse this baseline in order to control its development process as far as faults are concerned. An example of off-line tailoring is the adaptation of a general scientific paradigm such as "divide and conquer" to the software engineering domain.

- **On-line tailoring and generalizing** is concerned with tailoring candidate process and product-related experience to the specific needs and characteristics of a project and the chosen software evolution environment. These activities are not explicitly reflected in Figure 2 because they are integral part of the (re-)use activity. An example of on-line tailoring is the adaptation of a design inspection method to better detect the fault types anticipated in the current project [6]. An example of on-line generalization is the inclusion of project specific effort data from a past project into the domain specific effort baseline in order to better plan the required resources for the current project. Obviously, this kind of generalization could have been performed off-line too.

It is important to find a cost-effective balance between off-line and on-line tailoring and generalization. It can be expected that generalization is predominantly performed off-line, tailoring on-line.

A good developer is capable of informally tailoring general and domain specific experience to the specific needs of his or her project. Performing these transformations on existing experience assumes the ability to generalize experience to a broader context than the one studied, or to tailor experience to a specific project. The better this experience is packaged, the better our understanding of the environment. Maintaining a body of experience acquired during a

number of projects is one of the prerequisites for learning and feedback across projects.

A misunderstanding of the importance of tailoring exists in many organizations. These organizations have specific development guidebooks which are of limited value because they "are written for some ideal project" which "has nothing in common with the current project and, therefore, do not apply" [23]. All guidebooks (including standards such as DOD-STD-2167) are general and need to be tailored to each project in order to be effective.

3.2.3. Formalizing Existing Experience Prior to its Potential Reuse

The objective of formalizing existing experience prior to its potential reuse is to increase the reuse potential of a candidate reuse object by encoding it in more precise, better understood ways. This requires models of the various reuse objects, notations for making the models more precise, notations for abstracting reuse object characteristics, mechanisms for validating these models, and mechanisms for interpreting models in the appropriate context. Formalization activities are concerned with movement across the boundaries of the formality dimension within the experience base: from informal to schematized and then to productized. These activities are labeled with "F" in Figure 2.

Examples of formalizing experience include such activities as (a) writing functional specifications for a code module, (b) turning a lessons learned document into a management system that supports decision making, (c) building a cost model empirically based upon the data available, (d) developing evaluation criteria for evaluating the performance of a particular method, or (e) automating methods into tools.

In the SEL, measurement data have been explicitly formalized into cost models [1] and error models enabling the better planning and control of software projects with regard to cost estimation and the effectiveness of fault detection and isolation methods [2, 6, 8, 18]. Lessons learned have been integrated into expert systems aimed at supporting the management decision process [5, 24].

The more we can formalize experience, the better it can be reused. Therefore, we try not only to record experience, but over time to formalize experience from entirely informal (e.g., concepts), to structured or schematized (e.g., methods), or even to completely formal (e.g., tools). The potential for misunderstanding or misinterpretation decreases as experience is described more formally. To the same degree the experience can be modified more easily, or in the case of processes, it may be executed automatically (e.g., tools) rather than manually (e.g., methods).

3.2.4. (Re–) Using Existing Experience

The objective of reusing existing experience is to maximize the effective use of previously recorded experience during the planning and execution of all projects within an organization. This requires a precise characterization of the available candidate reuse objects, a precise characterization of the reuse–enabling environment including the evolution process that is expected to enable reuse, and mechanisms that support the reuse of experience. We must support the (re–)use of existing experience during the specification of reuse needs in order to compare them with descriptions of existing experience, the identification and understanding of candidate, the evaluation of candidate reuse objects, the possible tailoring of the reuse object, the integration of the reuse object into the ongoing software project, and the evaluating of the project's success. All information flows between the experience base and the software evolution process reflecting the (re–)use of experience are labeled with "U" in Figure 2.

Examples of reusing experience include such activities as (a) using code components from the repository, (b) developing a risk management plan based upon the lessons learned from applying a new technology, (c) estimating the cost of a project based on data collected from past projects, or (d) using a development method created for a prior project.

In the SEL, reuse needs are informally specified as part of the requirements document. Matching candidate requirements and design documents are identified by managers who are experienced in this environment. The evaluation of those candidate reuse objects is in part based

on human experience and in part on measurement data. They are tailored based on the application-domain knowledge of the personnel. They are integrated into a very stable evolution process based on human experience. All this reuse is implicit except for the reuse of code, which although explicit, is informal. It could only be successful because it evolved within a very stable environment. The recent change from FORTRAN to Ada has resulted in drastic changes of this environment and as a consequence to the loss in the implicit reuse heritage.

Since the key for improvement of products is always improvement of the process creating those products, we need to put equal emphasis on the reuse of product and process oriented experience. Even today, we have examples of reuse of process experience such as process plans (standards such as DOD-STD-2167, management plans, schedules) or process data (error, effort or reliability data that define baselines regarding software evolution processes within a specific organization). In most of these cases the actual use of this information within a specific project context is not supported; it is up to the respective manager to find the needed information, and to make sense out of it in the context of the current project.

4. TAME: AN INSTANTIATION OF THE REUSE-ENABLING ENVIRONMENT MODEL

The objective of the reuse-enabling software evolution environment model of Section 3.2 is to explicitly model the learning and reuse-related activities of recording experience, generalizing and tailoring experience, formalizing experience, and (re-)using experience so that they can be understood, evaluated, predicted and motivated.

In order to instantiate a specific reuse-enabling environment, we need to choose a model of the software evolution process itself. In general, such an evolution process model needs to be capable of describing the integration of learning and reuse into the software evolution process. In particular, it needs to be capable of modeling when experience is created and recorded into the

experience base as well as when existing experience is used. It needs to provide analysis for the purpose of on-line feedback, evaluating the application of all reuse experience, and off-line feedback for improving the experience base.

The reuse-enabling TAME environment model depicted in Figure 3 is an instantiation of the reuse-enabling software environment model of Section 3.2. based on a very general improvement oriented evolution process model.

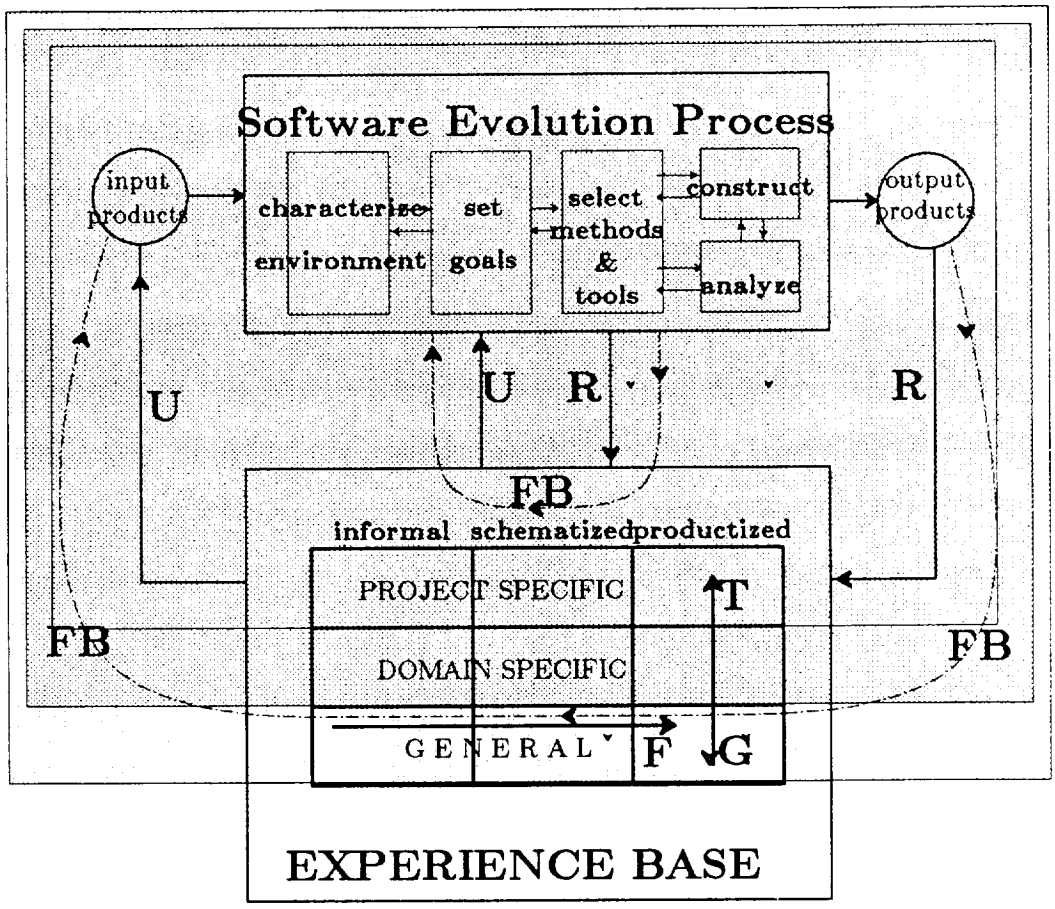


Figure 3: Reuse-Enabling "TAME" Environment Model

Each software project performed according to this improvement oriented evolution process model consists of a planning and an execution stage. The planning stage includes a characteriza-

tion of the current status of the project environment, the setting of project and improvement goals, and the selection of construction and analysis methods and tools that promise to meet the stated goals in the context of the characterized environment. The execution stage includes the construction of output products and the analysis of these construction processes and resulting output products.

The TAME environment model gives us a basis for discussing the integration of the recording and (re–)use activities into the software evolution process. During the environment characterization stage of the improvement oriented process model we (re–)use knowledge about the needs and characteristics of previous projects and record the needs and characteristics of the current project into the experience base. During the goal setting stage we (re–)use existing plans for construction and analysis from similar projects and record the new plans which have been tailored to the needs of the current project into the experience base. During the method and tool selection stage, we (re–)use as many of the constructive and analytic methods and tools which had been used successfully in prior projects of similar type as feasible and record possibly tailored versions of these methods and tools into the experience base. During construction we apply the selected methods and tools, and record the constructed products into the experience base. During analysis we use the selected methods and tools in order to collect and validate data and analyze them, and record the data, analysis results and lessons learned into the experience base.

The TAME environment explicitly supports the capturing of all kinds of experience. The consistent application of the improvement oriented process model across all projects within an organization provides a mechanism for evaluating the recorded experience, helping us to decide what and how to reuse, tailoring and analyzing. TAME supports continuous learning. The explicit and comprehensive modeling of the reuse-enabling evolution environment including the experience base, the evolution process, and the various learning and reuse activities (see Figure 3) allows us to measure and evaluate all relevant aspects of reuse. The measurement methodology used and supported within the TAME environment has been published in earlier papers [7, 8].

5. CONCLUSIONS

In this paper we have motivated and outlined the scope of a comprehensive reuse framework, introduced a reuse-enabling software environment model as a first step towards such a comprehensive reuse framework, and presented a first instantiation of such an environment in the context of the TAME (Tailoring A Measurement Environment) project at the University of Maryland [7, 8].

The reuse-enabling software evolution environment model presented in Section 3 provides a basic environment for supporting the recording of experience, the off-line generalization and tailoring of experience, the off-line formalization of experience, and the (re-) use of existing experience.

Further steps required towards the outlined reuse framework are more specific models of each of these activities that differentiate the components of these activities and serve as a basis for characterization, discussion and analysis. We are currently taking the reuse-enabling software environment model of section 3.2 down one level and developing a model for (re-)using experience. Based on this reuse model we will develop a reuse taxonomy allowing for the characterization of any instance of reuse. The reuse model will provide insight into the other activities of the reuse-enabling environment model only in the way they interact with the (re-)use activity. Corresponding models for each of the other activities need to be developed and integrated into the reuse-enabling software environment model.

The reuse-enabling TAME environment model serves as a basis for better understanding, evaluating and motivating reuse practices and necessary research activities. Performing projects according to the TAME environment model requires powerful automated support for dealing with the large amounts of experience and performing the complicated activities of recording, generalizing and tailoring, formalizing, and (re-)using experience. Indispensable components of such an automated support system are a powerful experience base, and a measurement support system. Many of the reuse approaches in the past have assumed that the developer has sufficient implicit

knowledge of the characteristics of the particular project environment, specific needs for reuse, the candidate reuse objects, etc. It is not trivial to have all this information available. The institutionalized learning of an organization and the proper documentation of that knowledge is definitely one of the keys to effective reuse. This leads to even better specification methods and tools (one of the frequently mentioned keys to effective reuse).

As part of the TAME project at the University of Maryland we have been working on providing appropriate support for building such an experience base, and supporting learning and (re-)use via measurement. We have completed several components towards a first prototype TAME system. These components include the definition of project goals and their refinement into quantifiable questions and metrics, the collection and validation of data, their analysis, and the storage of all kinds of experience. One of the toughest research problems is to use measurement not only for analysis, but also for feedback (learning and reuse) and planning purposes. We need more understanding of how to support feedback and planning. The TAME system is intended to serve as a vehicle for our research towards the effective support of explicit learning and reuse as outlined in this paper.

6. ACKNOWLEDGEMENTS

We thank all our colleagues and graduate students who contributed to this paper by either working on the TAME or any other reuse-related project or reviewing earlier versions of this paper.

7. REFERENCES

[1] J. Bailey, V. R. Basili, A Meta-Model for Software Development Resource Expenditures," in Proc. Fifth International Conference on Software Engineering, San Diego, USA, March 1981, pp. 107-116.

[2] V. R. Basili, "Can We Measure Software Technology: Lessons Learned from Eight Years of Trying," in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1985.

[3] V. R. Basili, "Quantitative Evaluation of Software Methodology," Dept. of Computer Science, University of Maryland, College Park, TR-1519, July 1985 [also in Proc. of the First Pan Pacific Computer Conference, Australia, September 1986].

[4] Victor R. Basili, "Software Maintenance = Reuse-Oriented Software Development," in Proc. Conference on Software Maintenance, Key-Note Address, Phoenix, AZ, October 1988.

[5] V. R. Basili, C. Loggia Ramsey, "ARROWSMITH-P - A Prototype Expert System for Software Engineering Management," IEEE Proceedings of the Expert Systems in Government Symposium, McLean, VA, October 1985, pp. 254-264.

[6] V. R. Basili, H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," Proc. of the Ninth International Conference on Software Engineering, Monterey, CA, March 30 - April 2, 1987, pp. 345-357.

[7] V. R. Basili, H. D. Rombach, "TAME: Integrating Measurement into Software Environments," Technical Report TR-1764 (or TAME-TR-1-1987), Dept. of Computer Science, University of Maryland, College Park, MD 20742, June 1987.

[8] V. R. Basili, H. D. Rombach "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758-773. [is also available as Technical Report (UMIACS-TR-88-8, CS-TR-1983, or TAME-TR-2-1988), Department of Computer Science, University of Maryland, College Park, MD 20742].

[9] V. R. Basili, H. D. Rombach, J. Bailey, and B. G. Joo, "Software Reuse: A Framework," Proc. of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987.

[10] V. R. Basili, R. W. Selby, D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, vol.SE-12, no.7, July 1986, pp.733-743.

[11] V. R. Basili and M. Shaw, "Scope of Software Reuse," White paper, working group on 'Scope of Software Reuse', Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987 (in preparation).

[12] Ted Biggerstaff, "Reusability Framework, Assessment, and Directions," IEEE Software Magazine, March 1987, pp.41-49.

[13] P. Freeman, "Reusable Software Engineering: Concepts and Research Directions," Proc. of the Workshop on Reusability, September 1983, pp. 63-76.

[14] R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability," IEEE Software, vol.4, no.1, January 1987, pp. 6-16.

[15] IEEE Software, special issue on 'Reusing Software', vol.4, no.1, January 1987.

[16] IEEE Software, special issue on 'Tools: Making Reuse a Reality', vol.4, no.7, July 1987.

[17] G. A. Jones, R. Prieto-Diaz, "Building and Managing Software Libraries," Proc. Compsac'88, Chicago, October 5-7, 1988, pp. 228-236.

[18] F. E. McGarry, "Recent SEL Studies," in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, Dec. 1985.

[19] Mary Shaw, "Purposes and Varieties of Software Reuse," Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July, 1987.

[20] T. A. Standish, "An Essay on Software Reuse," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984, pp.494-497.

THE VIEWGRAPH MATERIALS

FOR THE

V. BASILI PRESENTATION FOLLOW

TOWARD A REUSE-ORIENTED SOFTWARE EVOLUTION PROCESS

VICTOR R. BASILI
H. DIETER ROMBACH

INSTITUTE FOR ADVANCED COMPUTER STUDIES
AND
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

REUSE OF EXPERIENCE IS THE KEY TO PRODUCTIVITY AND QUALITY

EXPERIENCE INCLUDES PRODUCTS, PROCESSES AND KNOWLEDGE

MOST REUSE IS AD HOC, IMPLICIT, AT CODE LEVEL

REUSE MUST BE BUILT INTO THE PROCESS

MODELS OF REUSE-ORIENTED EVOLUTION ENVIRONMENT AND ACTIVITIES MUST BE DEVELOPED

IMPROVEMENT PARADIGM

- **CHARACTERIZE the current project environment**
- **SET UP GOALS and REFINE THEM INTO QUANTIFIABLE QUESTIONS AND METRICS for successful project performance and improvement over previous project performances**
- **CHOOSE the appropriate construction model for this project and supporting methods and tools**
- **EXECUTE the processes and construct the products, collect the prescribed data, validate it, and provide feedback in real-time**
- **ANALYZE the data to evaluate the current practices, determine problems, record the findings and MAKE RECOMMENDATIONS FOR IMPROVEMENT**
- **Proceed to step 1 to START THE NEXT PROJECT, ARMED WITH THE EXPERIENCE GAINED FROM THIS AND PREVIOUS PROJECTS**

REUSE IN THE SEL

IMPLICIT/THROUGH PEOPLE

- APPLICATION DOMAIN
- SOLUTION STRUCTURE
- MANAGEMENT/SUPPORT

EXPLICIT/THROUGH PROCESS

- CODE REUSE

QUESTIONS:

- WHAT HAPPENS TO REUSE AS WE MOVE FROM FORTRAN TO ADA?
- HOW DO WE MEASURE THE EFFECTS OF REUSE?
- WHAT IS THE EFFECT OF REUSE ON ALL ASPECTS OF THE LIFE CYCLE?

TRADITIONAL SOFTWARE EVOLUTION

TYPICALLY SEE'S

PROVIDE THE PROJECT DATA BASE

SUPPORT THE INTERACTION OF PEOPLE WITH METHODS,
TOOLS AND THE PROJECT DATA BASE

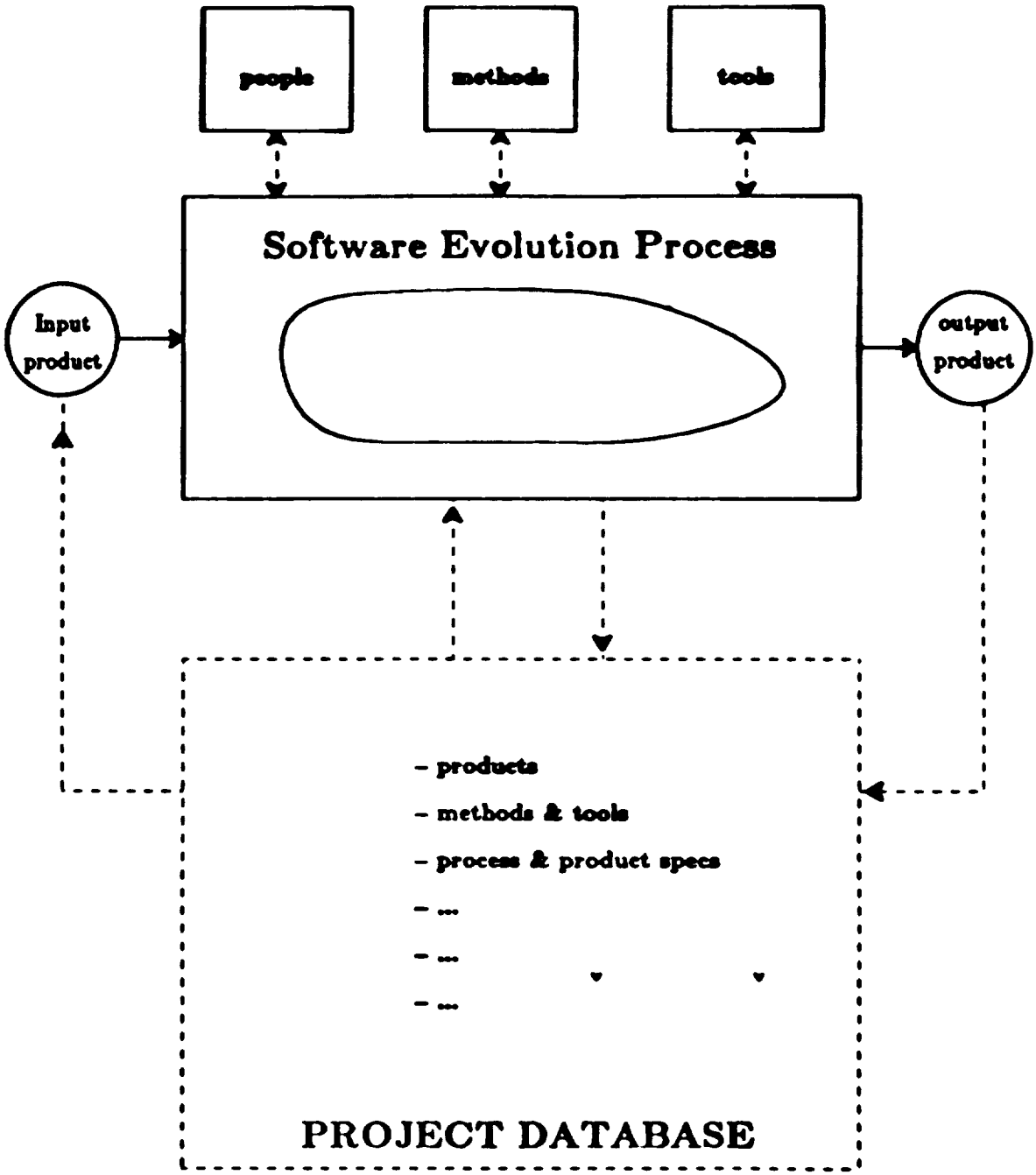
EXPERIENCE IS NOT

CONTROLLED BY THE PROJECT DATA BASE

OWNED BY THE ORGANIZATION

REUSE EXISTS IMPLICITLY

TRADITIONAL SE MODEL



A REUSE-ORIENTED EVOLUTION ENVIRONMENT MODEL

WHAT ARE THE COMPONENTS OF A REUSE-ORIENTED EVOLUTION MODEL?

HOW CAN THE REUSE PROCESS MODEL BE INCORPORATED INTO THE CONTEXT OF DEVELOPMENT AND MAINTENANCE?

HOW CAN LEARNING AND FEEDBACK BE USED TO SUPPORT THE REUSE MODEL?

DEFINITIONS

IMPROVEMENT

ENHANCING A SOFTWARE PROCESS OR PRODUCT WITH RESPECT TO QUALITY OR PRODUCTIVITY

FEEDBACK

RETURNING TO THE ENTRY POINT OF SOME PROCESS ARMED WITH THE EXPERIENCE GAINED FROM PREVIOUS PERFORMANCES OF THIS PROCESS

LEARNING

THE ACTIVITY OF ACQUIRING KNOWLEDGE BY INSTRUCTION, E.G., CONSTRUCTION, OR STUDY, E.G., ANALYSIS

REUSE

THE ACTIVITY OF REPEATEDLY USING EXISTING EXPERIENCE, AFTER RECLAIMING IT, WITH OR WITHOUT MODIFICATION

EXPERIENCE BASE

A REPOSITORY OF ALL KINDS OF EXPERIENCE

RELATIONSHIP OF THE TERMS

IMPROVEMENT OF A SOFTWARE PROCESS OR PRODUCT

REQUIRES THE FEEDBACK OF AVAILABLE EXPERIENCE INTO SOME PROCESS

FEEDBACK

REQUIRES THE

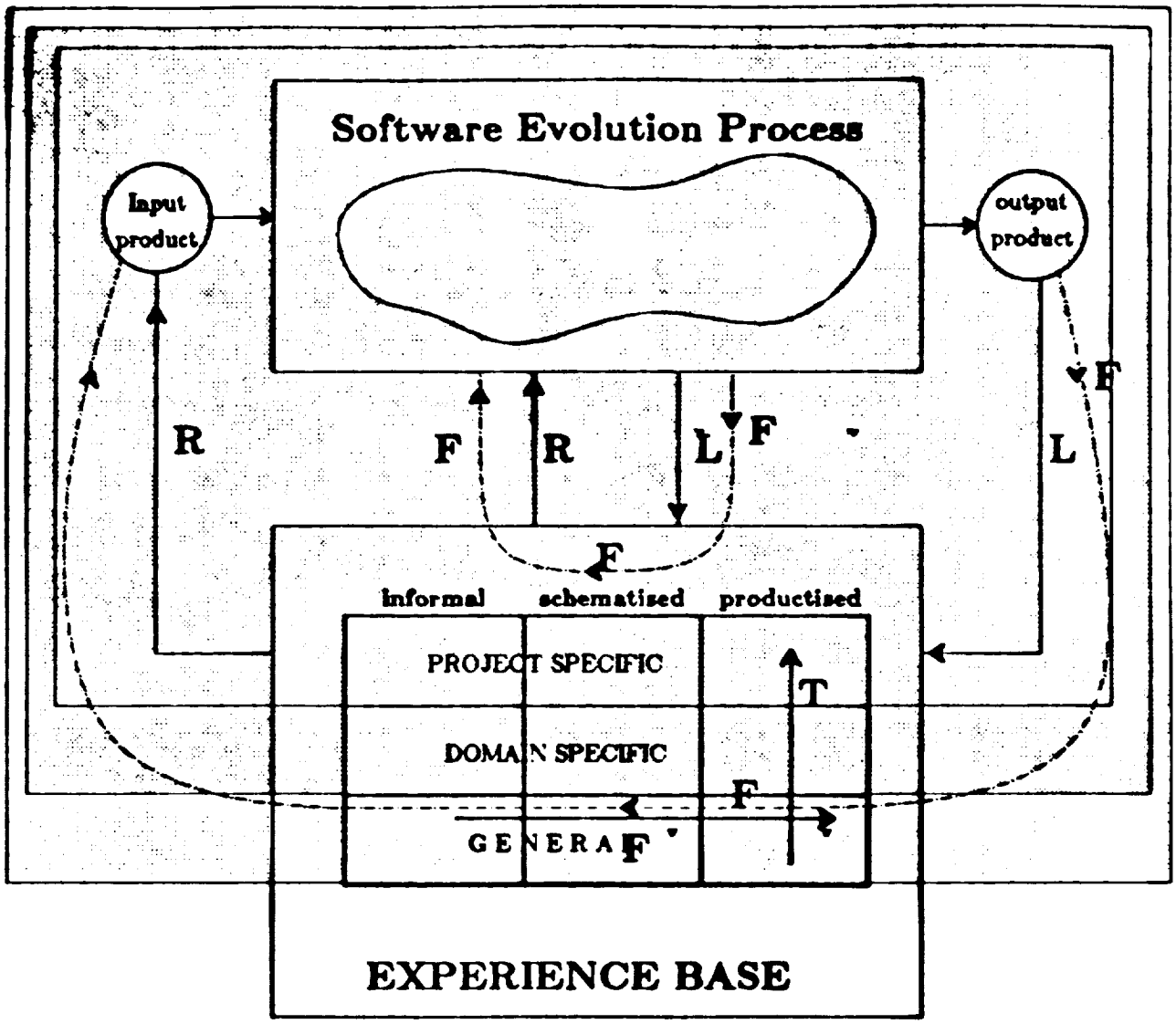
ACCUMULATION OF EXPERIENCE (LEARNING)

INTO SOME AVAILABLE RESOURCE (EXPERIENCE BASE)

THE USE OF THIS EXPERIENCE FOR A PARTICULAR PURPOSE (REUSE)

EXPERIENCE BASES CAN BE DATA BASES, INFORMATION BASES, KNOWLEDGE BASES OR ANY COMBINATION OF THE THREE

RE-USE ORIENTED SE MODEL



SYSTEMATIC LEARNING AND REUSE

SYSTEMATIC LEARNING REQUIRES SUPPORT FOR

- RECORDING EXPERIENCE
- OFF-LINE GENERALIZING OR TAILORING OF EXPERIENCE
- FORMALIZING OF EXPERIENCE

SYSTEMATIC REUSE REQUIRES SUPPORT FOR

- USING EXISTING EXPERIENCE
- ON-LINE GENERALIZING OR TAILORING OF CANDIDATE EXPERIENCE

BOTH LEARNING AND REUSE NEED TO BE INTEGRATED INTO AN OVERALL SOFTWARE EVOLUTION MODEL

RECORDING EXPERIENCE

OBJECTIVE:

CREATE A REPOSITORY OF WELL-SPECIFIED AND CLASSIFIED EXPERIENCE

REQUIREMENTS:

EFFECTIVE MECHANISMS FOR COLLECTING, VALIDATING, STORING AND RETRIEVING EXPERIENCE

EXAMPLES:

STORING OF CODE COMPONENTS FROM PRIOR SYSTEMS IN A REPOSITORY, APPROPRIATELY DOCUMENTED, CATALOGED AND CATEGORIZED

CATALOGING OF A SET OF LESSONS LEARNED IN APPLYING A NEW TECHNOLOGY

SAVING MEASUREMENT DATA IN A DATA BASE ON THE COST OF DEVELOPING A SYSTEM

RECORDING A DEVELOPMENT METHOD FOR USE ON THE NEXT PROJECT

(RE-)USING EXISTING EXPERIENCE

OBJECTIVE:

MAXIMIZING THE EFFECTIVE USE OF PREVIOUSLY RECORDED EXPERIENCE DURING THE PLANNING AND EXECUTION OF ALL PROJECTS WITHIN AN ORGANIZATION

REQUIREMENTS:

- SPECIFICATION OF THE REUSE ENVIRONMENT
- CHARACTERIZED CANDIDATE REUSE OBJECTS
- AVAILABLE EXPERIENCE
- A PROCESS IN WHICH WE
  - SPECIFY REUSE NEEDS
  - FIND APPROPRIATE CANDIDATES
  - EVALUATE REUSE CANDIDATES
  - MODIFY THE REUSE CANDIDATE
  - INTEGRATE THE REUSE CANDIDATE INTO THE PROCESS
  - TEST THE INTEGRATED OBJECT WHICH INCLUDES THE REUSE OBJECT

EXAMPLES:

- USING CODE COMPONENTS FROM THE REPOSITORY
- DEVELOPING A RISK MANAGEMENT PLAN BASED UPON LESSONS LEARNED IN APPLYING A NEW TECHNOLOGY
- ESTIMATING THE COST OF A PROJECT USING DATA ON PAST PROJECTS
- USING A DEVELOPMENT METHOD CREATED FOR A PRIOR PROJECT

GENERALIZING OR TAILORING OF EXISTING EXPERIENCE PRIOR TO ITS REUSE

OBJECTIVE: GENERALIZING

MAKING A CANDIDATE REUSE OBJECT USEFUL IN A LARGER SET OF POTENTIAL TARGET APPLICATIONS

OBJECTIVE: TAILORING

FINE-TUNING A CANDIDATE REUSE OBJECT TO FIT A SPECIFIC TASK OR EXHIBIT SPECIAL ATTRIBUTES, SUCH AS SIZE OR PERFORMANCE

NOTE:

GENERALIZING AND TAILORING CAN BE ON-LINE OR OFF-LINE

- ON-LINE: DONE FOR A SPECIFIC PROJECT
- OFF-LINE: THE PRECISE REUSE CONTEXT NOT KNOWN A PRIORI

REQUIREMENTS:

A WELL-DOCUMENTED CATALOGED AND CATEGORIZED SET OF REUSE OBJECTS

MECHANISMS FOR EASY MODIFICATION

AN UNDERSTANDING OF THE POTENTIAL TARGET APPLICATIONS

EXAMPLES:

DEVELOPMENT OF A GENERIC PACKAGE FROM A SPECIFIC PACKAGE

INSTANTIATING A GENERIC PACKAGE FOR A SPECIFIC DATA TYPE

GENERALIZING THE LESSONS LEARNED FROM A SPECIFIC DESIGN TECHNOLOGY FOR A SPECIFIC APPLICATION TO ANY DESIGN FOR THAT APPLICATION OR ANY APPLICATION

PARAMETERIZING A COST MODEL FOR A SPECIFIC ENVIRONMENT

MODIFYING THE DESIGN INSPECTION PROCESS BASED UPON A HISTORY OF THE DEFECTS MADE IN THE SPECIFIC ENVIRONMENT

FORMALIZATION OF EXPERIENCE

OBJECTIVE:

THE ENCODING OF EXPERIENCE IN MORE PRECISE, BETTER UNDERSTOOD WAYS

REQUIREMENTS:

MODELS OF VARIOUS REUSE OBJECTS

NOTATIONS FOR MAKING THE MODELS MORE PRECISE

NOTATIONS FOR ABSTRACTING REUSE OBJECT CHARACTERISTICS

MECHANISMS FOR VALIDATING THE MODELS

MECHANISMS FOR INTERPRETING MODELS IN CONTEXT

EXAMPLES:

WRITING THE FUNCTIONAL SPECIFICATION OF A CODE MODULE

TURNING A LESSONS LEARNED DOCUMENT INTO A MANAGEMENT SYSTEM
THAT SUPPORTS DECISION MAKING

BUILDING A COST MODEL EMPIRICALLY BASED UPON DATA AVAILABLE

DEVELOPING EVALUATION CRITERIA FOR EVALUATING THE PERFORMANCE
OF A PARTICULAR METHOD

AUTOMATING METHODS INTO TOOLS

INTEGRATION OF REUSE AND LEARNING INTO A SOFTWARE EVOLUTION PROCESS MODEL

OBJECTIVE:

TO SUPPORT THE LEARNING AND REUSE PROCESSES IN A WELL-SPECIFIED, ORGANIZED, NATURAL WAY SO THAT IT CAN BE UNDERSTOOD, EVALUATED, PREDICTED AND MOTIVATED

REQUIREMENTS:

SUPPORT MECHANISMS FOR

- RECORDING WHAT HAS BEEN LEARNED
- (RE-)USING AND ON-LINE TAILORING OR GENERALIZING
- OFF-LINE TAILORING
- FORMALIZATION

EXAMPLES:

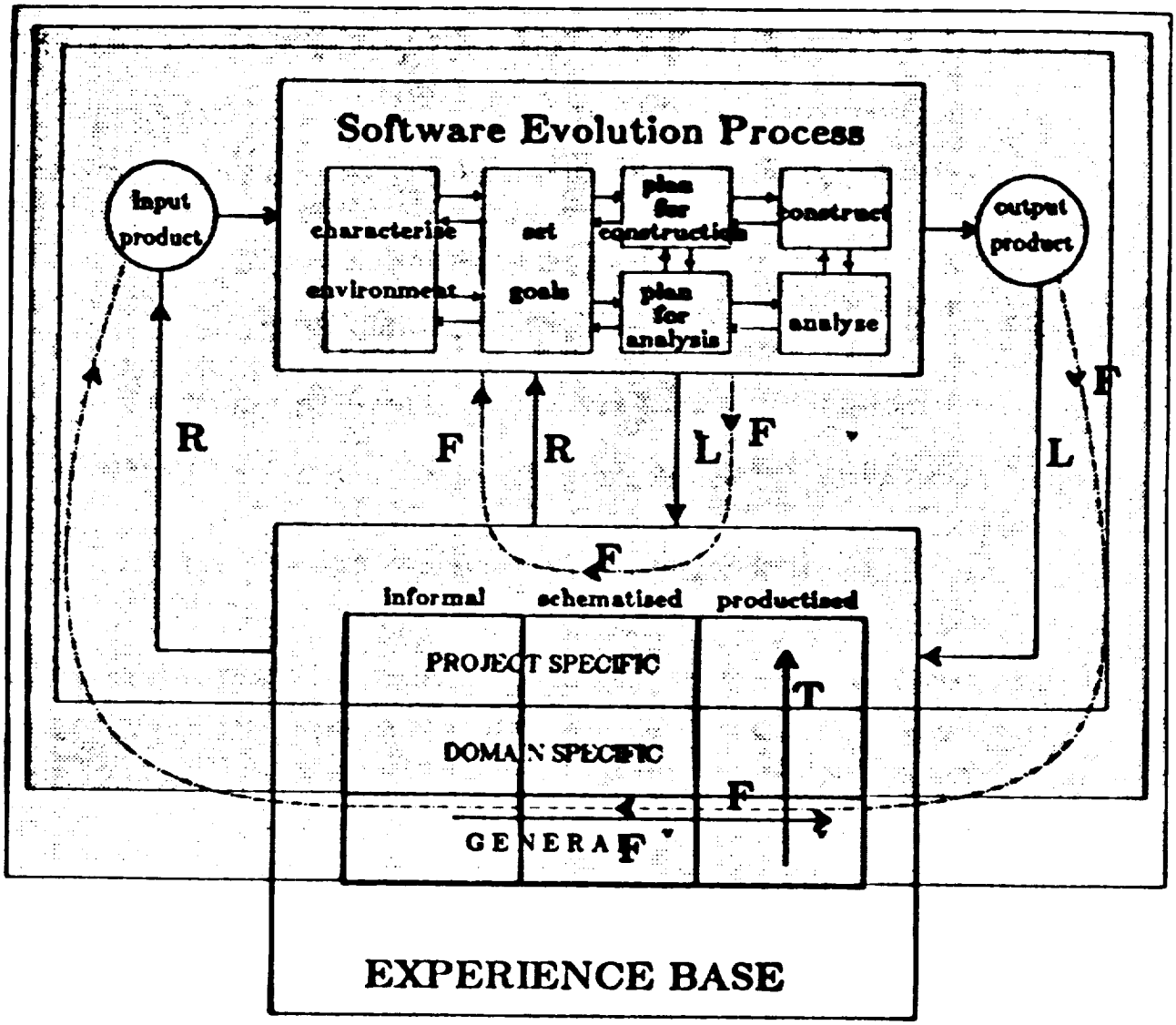
A REPOSITORY FOR ALL POSSIBLE CANDIDATE REUSE OBJECTS INCLUDING METHODS, TOOLS, PRIOR PROJECT DOCUMENTS (CODE, REQUIREMENTS, RISK MANAGEMENT PLANS)

A SET OF MODELS FOR VARIOUS PROCESSES AND PRODUCTS

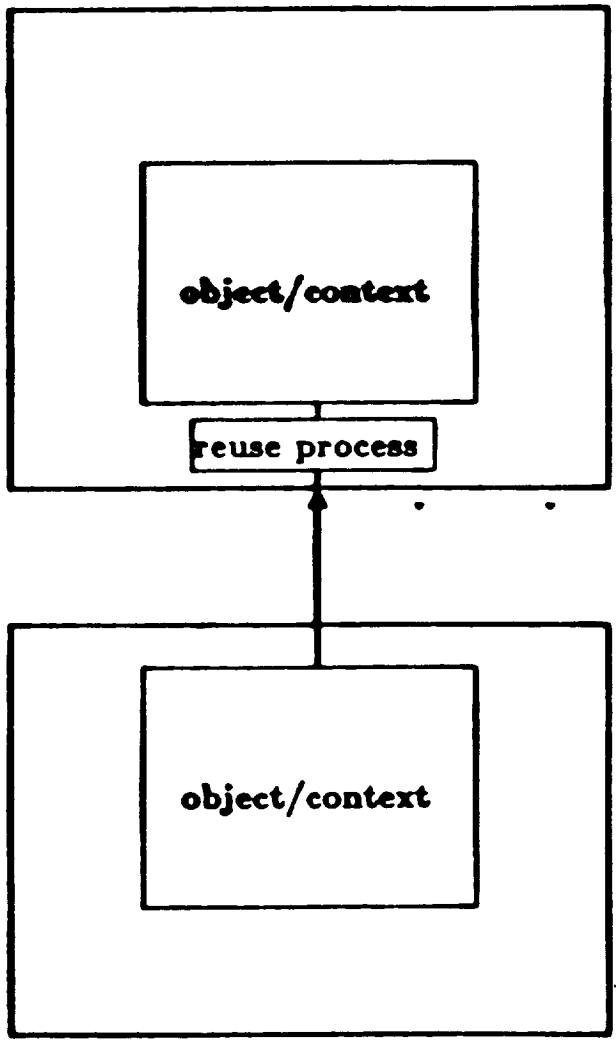
A MEASUREMENT DATA BASE

A KNOWLEDGE BASE THAT SUPPORTS MANAGEMENT DECISION-MAKING BASED UPON DATA, LESSONS LEARNED AND OTHER AVAILABLE INFORMATION

IMPROVEMENT RE-USE ORIENTED SE MODEL

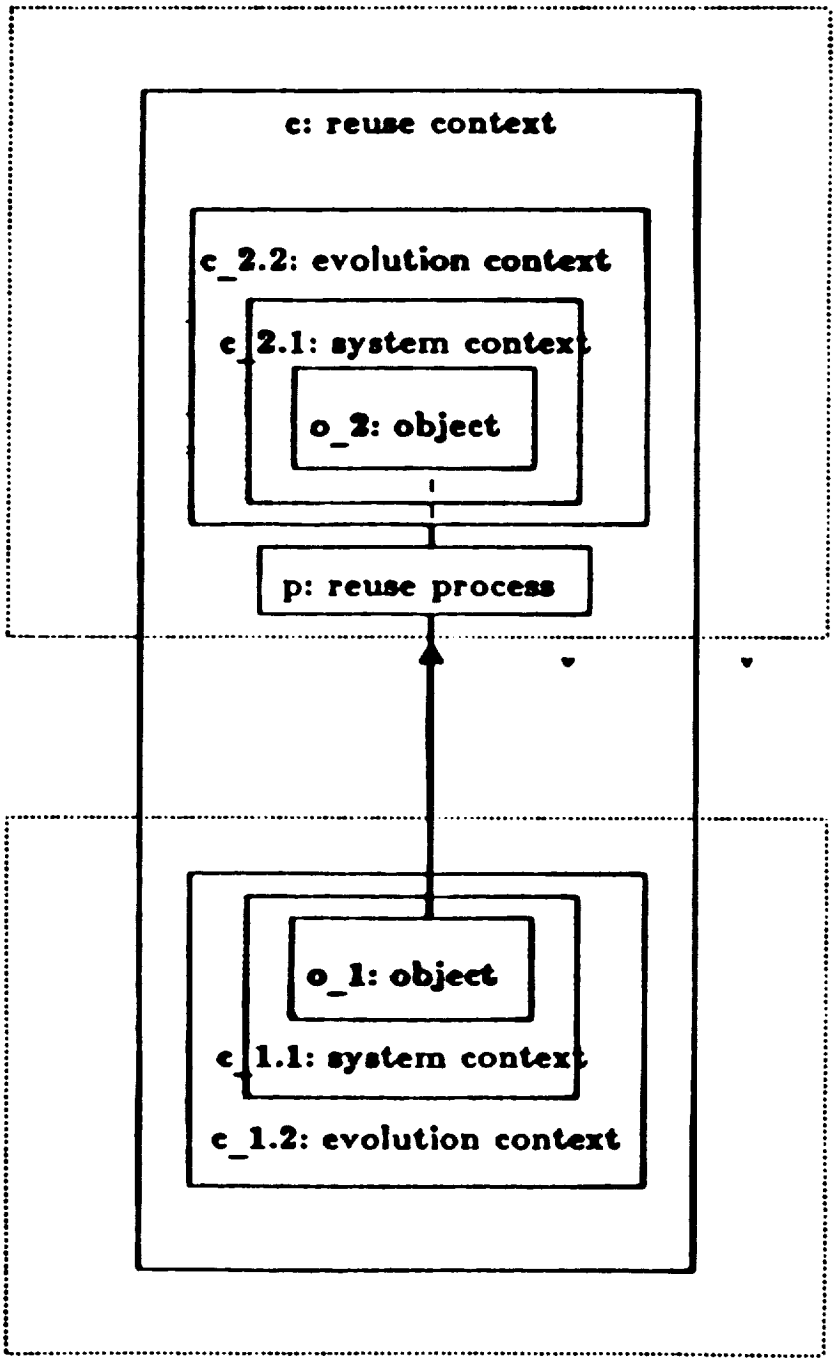


REUSE–ENABLING SOFTWARE EVOLUTION PROCESS



EXPERIENCE BASE

REUSE–ENABLING SOFTWARE EVOLUTION PROCESS



EXPERIENCE BASE

CONCLUSIONS

GENERAL

NEED INTEGRATED MODELS OF ALL THE ACTIVITIES;
E.G., BALANCE BETWEEN REUSE AND TAILORING

NEED TO USE MODELS AND PROJECT GOALS TO DEVELOP USEFUL MEASURES

GOALS AND EFFECTS OF REUSE MUST BE EXPLICITLY STATED SO WE CAN CHARATERIZE, EVALUATE, PREDICT AND MOTIVATE REUSE

SEL

MOVING TO ADA (OR ANY NEW TECHNOLOGY) COSTS IN THE SHORT RUN, BUT AN EXPLICIT REUSE CHARACTERIZATION CAN HELP

EFFECT IS MORE THAN LINES OF CODE REUSED

ARE MOVING TOWARD BUILDING AN EXPERIENCE BASE TO SUPPORT TAILORING AND REUSE

The Software Management Environment (SME)

Jon D. Valett
(NASA/GSFC)
William Decker
and
John Buell
(Computer Sciences Corporation)

N91-10609

1.0. Background (charts 1 and 2)

The Software Management Environment (SME) is a research effort designed to utilize the past experiences and results of the Software Engineering Laboratory (SEL) [Card82] and to incorporate this knowledge into a tool for managing projects. SME provides the software development manager with the ability to observe, compare, predict, analyze, and control key software development parameters such as effort, reliability, and resource utilization. This paper describes the major components of the SME, outlines the architecture of the system, and provides examples of the functionality of the tool.

The SEL has been researching and evaluating software development methodologies for over ten years. This research has provided valuable insight into the software development process of one particular organization. By collecting detailed software development data and recording that data in a software engineering data base [Church82][Heller87], the SEL has been able to characterize and understand the development process within that organization. Using this data to measure the impact of various methodologies, tools, and perturbations to that process has enabled the SEL to better control and manage the software projects of this organization.

Recognizing the vast potential of providing the experience of previous projects, the data, the research results, and the knowledge of experienced software managers to the managers of ongoing projects, research efforts were initiated to provide these items in the form of a tool. Initial prototype efforts began in 1984, with the development of a tool that explored the possibilities of providing this information. That effort was thoroughly analyzed and requirements were developed for a more complete software system late in 1986 [Valett87]. During this time work began on the current SME.

The major functionality that the SME provides for its user can be divided into four high level concepts:

1.) The ability for a manager to compare the ongoing software project to other projects. This function allows the manager to view software metric data such as weekly effort or error data and to compare it to other projects.

2.) The ability for the manager to receive predictions of future events of interest. SME will predict the final values for key project parameters such as effort or reliability.

3.) SME will also analyze project data to give insights into the strengths and weaknesses of the development process.

4.) SME will analyze overall project quality. This will

provide the manager with high-level insight into the project's overall development process.

Thus, the SME enables the manager to gain valuable insight into the progress and quality of a software development project.

This paper describes the concepts and architecture of the SME. Section 2.0 is devoted to describing the research results and data which are incorporated into the SME. Section 3.0 describes the architecture of the system and gives examples of the functions available to the manager. Finally, a brief discussion is presented in section 4.0.

2.0 The Components of SME

Attempting to integrate past research results along with dynamic project data, the SME provides the manager with a wide variety of information for monitoring and controlling an ongoing software project. The information required to provide this functionality can be broken into three major components: 1) the corporate history, 2) research results from studies of the software development process, and 3) management rules for software development.

2.1 The Corporate History (charts 3 and 4)

One underlying assumption of the SME is that a corporate history of some type exists. In this case, the SEL data base serves as the corporate memory for the SME. The SEL data base has evolved into its current form over the nearly 12 years of its existence. The data base itself provides the SME with the majority of the raw data required to monitor a project.

The major items of data provided by the data base include weekly software parameters that are of interest to the software manager. These weekly items of data include such parameters as effort, computer utilization, growth of source code, change history, and error history. All of these items are available as part of the SEL data base for any project of interest, as well as on the past projects that a manager may want to use as a basis for comparison.

Many of the other data needed by the SME is acquired from the SEL data base. This data includes items which characterize the types of projects as well as the language or tools used. Subjective data which is used to evaluate projects on a series of software methodology questions is also used by the system.

During the 12 years of the SEL's existence, numerous studies and reports characterizing and evaluating the software development environment have been written. These studies and reports have provided numerous research results for the environment. Thus, the SEL data base establishes the foundation for all of the components of the SME.

2.2 Research Results (chart 5)

A second major component of the SME is the research results that have been developed via the SEL data base. Information

derived from papers and studies developed through experimentation and through analysis of the SEL data base is a key part of the SME (for examples of results see [Valett88]). The SME attempts to incorporate these research results via models and measures for the software environment. Based on a comprehensive understanding of the development environment, these models and measures are used by the SME to enable the manager to better understand how a particular project compares to the normal project within the environment. They also are used by the SME in predicting and estimating future conditions on the software project.

Models of software development parameters are essential for the SME to perform its prediction and comparison functions. A model profiles the expenditure, the utilization, or the production of a software development parameter. As an example, a model of the staffing profile would capture the typical expenditure of effort over the entire software development life cycle [Basili78]. This type of model can be used by a manager to compare the current effort expenditure with the typical one for this environment.

Other types of relationships are used by the SME to capture known affects of specific software development methodologies. For instance, the knowledge that code reading is the most affective method for finding errors in this environment [Selby87], is important information to disseminate to a manager. One goal of the SME is to provide a knowledge base of known facts and relationships about a particular environment.

2.3 Software Development Rules (chart 6)

A final major component of the SME is software development rules. The SME attempts to integrate the experience of software managers into an expert system concept to provide the ability to analyze project measures and status. Previously, this experience was only captured in lessons learned or summary documents. The SME formalizes this knowledge into a basic structure that will continually evolve as the experience and knowledge are validated. By automating the knowledge utilization into an expert system, SME gives the manager the ability to apply past experience to current projects. The basic concept of utilizing expert systems for software management was proven feasible by previous research done by the SEL [Valett85][Ramsey86]. Admittedly, the extension of these concepts for use within the SME is an extremely difficult area of research, however, early results show they will be very useful.

Within the SME experienced manager's knowledge can be used in numerous areas. The knowledge has been collected from interviews with numerous managers, along with analysis of SEL data and information obtainable from the various reports and studies written by the SEL. An example of the type of knowledge used by the SME is shown in chart 6. This rule:

If error rate is lower than normal then
1. Insufficient testing
2. Experienced team
3. Problem less difficult than expected

is a simplified form of the type of rule collected for use in the

SME. Utilizing this rule, numerous other rules, and facts about the measures and status of the software project, the SME can reach conclusions pertaining to the deviations of project measures, such as error rate. Thus, the system can give the manager vital information regarding the strengths and weaknesses of a software development effort. In the future, this knowledge will also be used to provide the overall assessment functionality of the SME.

Obviously, the collection and validation of these rules and relationships is a major task. The research into this part of the SME will involve continual iteration and evolution. However, by establishing a baseline set of software rules and incorporating them into the SME and by constantly integrating feedback on the validity of the conclusions and knowledge, the SME knowledge base will mature into an even more valuable component of the system.

3.0 SME Architecture and Functionality

The SME architecture is designed to integrate the three major components described in section 2 into a tool which provides the manager with the functions of comparison, analysis, prediction, and expert guidance (see chart 7). The major processing of the system is performed on a VAX 11/780 and is written in Pascal, with the user interface and some data handling procedures performed on IBM/PC compatibles. The selection of this particular hardware architecture was driven by the desire to make SME accessible to managers in their offices and to provide color graphics capabilities. The remainder of this section is devoted to describing the major functionality of the SME: comparison, analysis, and prediction.

3.1 Comparison (charts 8 and 9)

The comparison function of the SME is designed to allow the manager to view project data on measures of interest such as effort, lines of code (LOC), CPU utilization, etc. and to compare these measures to past projects and to models of the normal project. Comparison utilizes the SEL data base and current project data along with models and measures of the typical project. Providing the comparison feature allows the manager to determine how the current project is behaving as it compares to past similar projects as well as whether or not the current project is following the "typical" pattern for that particular measure. In the examples chart 8 shows a comparison of the number of errors on a current project against the number errors on a past project, while chart 9 shows a similar comparison, except that the past project is replaced by a model of errors committed for the environment. These types of comparisons are available for a variety of project measures; they enable the manager to examine the characteristics of the current project in the context of other projects.

3.2 Analysis (chart 10)

Giving the user the knowledge of experienced software managers, the analysis function provides insights into the strengths and weaknesses of a project. Utilizing the SEL database, the current data, the models and measures, and the rule base, the analysis function compares the value for a certain measure for a current project to the model of that measure and reaches conclusions about why the project is deviating from the norm. The example shows a comparison of the number of errors on the current project with the model for errors. Since the number of errors is below what would be expected at this point in the software development, the SME can provide analysis as to why this condition may be occurring. The example illustrates a use of the rule discussed in section 2.3. While this is an elementary example, it does show the type of information SME provides. This type of analysis provides the manager with valuable insight into potential problems that might be occurring on the project of interest.

3.3 Prediction (chart 11)

Based on the current status of a software measure, the prediction function attempts to estimate the behavior of the measure through the completion of the project. Making heavy use of the models and measures along with the data for the project of interest, this function gives managers reasonable estimates of key project parameters. For example, given the current system size in LOC, information regarding the project's subjective profile, and some project estimates, SME predicts the final system size. Similarly, information on the current phase and error rate of a project along with certain models and measures, enables the SME to predict the final error rate for the system. Obviously, these and other key project parameters are invaluable to the manager in planning and controlling a software project.

4.0 Discussion (chart 12)

While the SME currently provides parts of all the capabilities described in section 3, it is still considered a research effort. Much research into each of the functions described as well as into other more advanced features of the system is still required for the system to become a fully useful tool. Thus, the system will change as these features are integrated into the overall architecture of the system.

In a similar manner, the system will continually evolve as the knowledge of the environment evolves. For example, although the current SME focuses on the waterfall life cycle model, as other paradigms are utilized and adopted within the environment, these results will be factored into the SME. The SME will continue to mature as long as research into the understanding of the development environment continues to provide an improved understanding of the software process.

Continuing to focus on utilizing the knowledge and experience of past research in addition to future research, the SME provides and will continue to provide a valuable feedback mechanism which encourages the reuse of this knowledge and

experience. The formalization of this reuse into a constantly maturing software tool, ensures that the knowledge will be captured and used on future software development efforts. Thus, the SME should continue to be a useful software management tool that will provide the software development manager with valuable information and insight into the quality of a software development project.

REFERENCES

[Basili78] Basili, V. and M. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, Vol. 10.

[Card82] Card, D., F. McGarry, G. Page, et al., The Software Engineering Laboratory, SEL-81-104, February 1982.

[Church82] Church, V., D. Card, and F. McGarry, Guide to Data Collection, SEL-81-101, August 1982.

[Heller87] Heller, G.,Data Collection Procedures for the Rehosted SEL Data Base, SEL-87-008, October 1987.

[Ramsey86] Ramsey, C. and V. Basili, "An Evaluation of Expert Systems for Software Engineering Management," TR-1708, University of Maryland, Technical Report, September 1986.

[Selby87] Selby, R. and V. Basili, "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, December 1987.

[Valett85] Valett, J. and A. Raskin, "DEASEL: An Expert System for Software Engineering," Proceedings of the Tenth Annual Software Engineering Workshop, SEL-85-006, December 1985.

[Valett87] Valett, J., "The Dynamic Management Information Tool (DYNAMITE): Analysis of Prototype, Requirements, and Operational Scenarios," Masters Thesis, The University of Maryland, May 1987.

[Valett88] Valett, J. and F. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," Proceedings of the 21st Annual Hawaii International Conferences on System Sciences, January 1988.

THE VIEWGRAPH MATERIALS

FOR THE

J. VALETT PRESENTATION FOLLOW

THE SOFTWARE MANAGEMENT ENVIRONMENT (SME)

JON VALETT
(NASA/GSFC)

BILL DECKER
JOHN BUELL
(CSC)

CHART 0

J. Valett
NASA/GSFC

SME CONCEPT

GOAL: To produce a tool (or set of tools) that utilizes historical information pertinent to software development in order to provide guidance and information for new development efforts

DRIVEN BY:

- Availability of
  - Archived data in the SEL
  - Results of SEL studies
  - Accumulation of expertise on the software development process
- Need for
  - Management guidance
  - Information availability
- Drive for research into improving the development process

SME GOALS

Provide the Software Development Manager with insights on an ongoing project via:

1. Comparison	• Compare development profile of current project with past projects
2. Prediction	• Predict future events (Cost/reliability...)
3. Analysis	• Determine strengths and weaknesses of project
4. Expert Guidance	• Determine overall quality of project

J. Valett
NASA/GSFC

C218.003

SEL PRODUCTION ENVIRONMENT

TYPE OF SOFTWARE: Scientific, ground based, interactive graphic
moderate reliability and response requirements

LANGUAGES: Primarily FORTRAN; some Ada and assembler

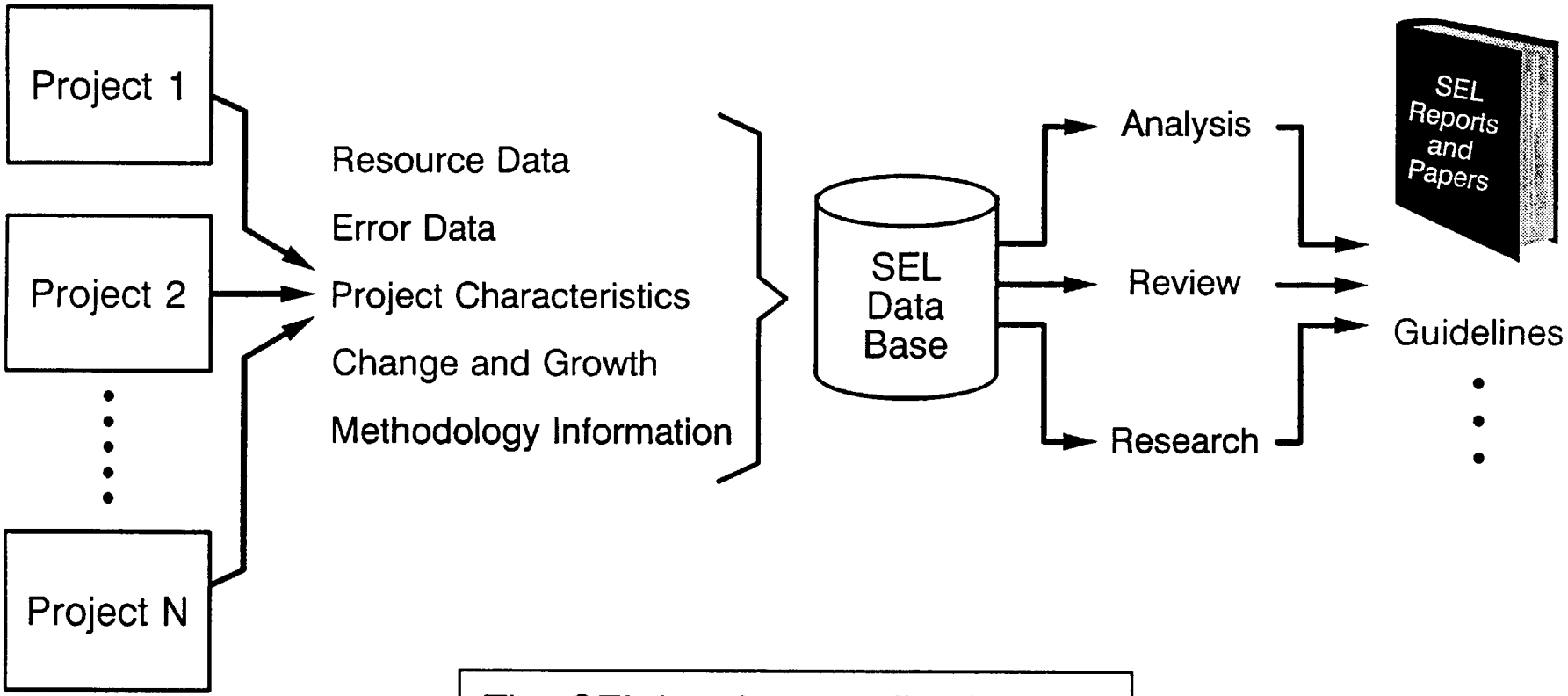
PROJECTS:
- Average duration 1-2 years (≈8 staff years)
- Average size 60-70 KSLOC (some projects 180-225 KSLOC)

EXPERIMENTATION:
- Monitor production level projects and collect data
- Apply methodologies and evaluate
- Apply results to future projects

In 1977, the SEL began collecting development data and studying the software development process. Since then over 65 projects have been monitored

C218.004

SEL Collects Software Development Data

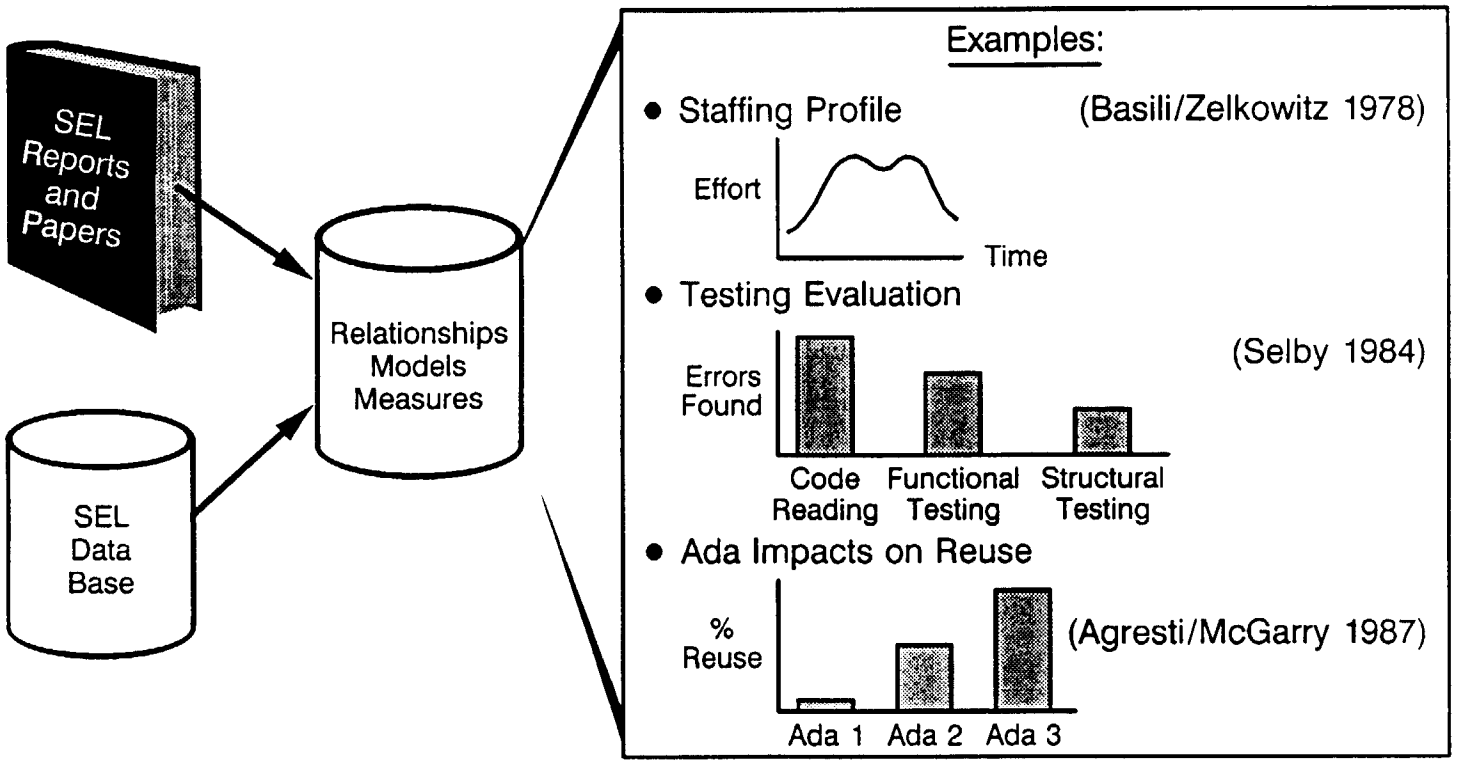


The SEL has been collecting and analyzing data for over 10 years.

J. Valett
NASA/GSFC

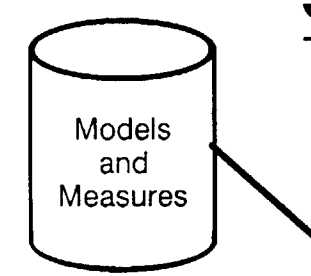
C218.005

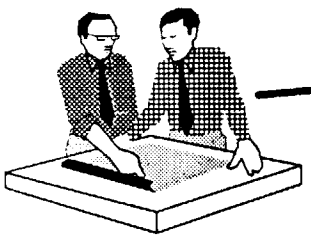
SEL RESEARCH RESULTS



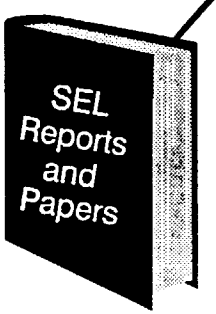
Studies of SEL projects and applied research have resulted in over 100 reports covering topics ranging from development profiles to "rules" of software development

SEL DEVELOPMENT OF SOFTWARE DEVELOPMENT RULES





Interviews with Experienced Managers



Analysis, Acquisition, Validation

Rules of Software Development

Example:

If error rate is * lower than normal then;

1. Insufficient testing
2. Experienced Team
3. Problem less difficult than expected

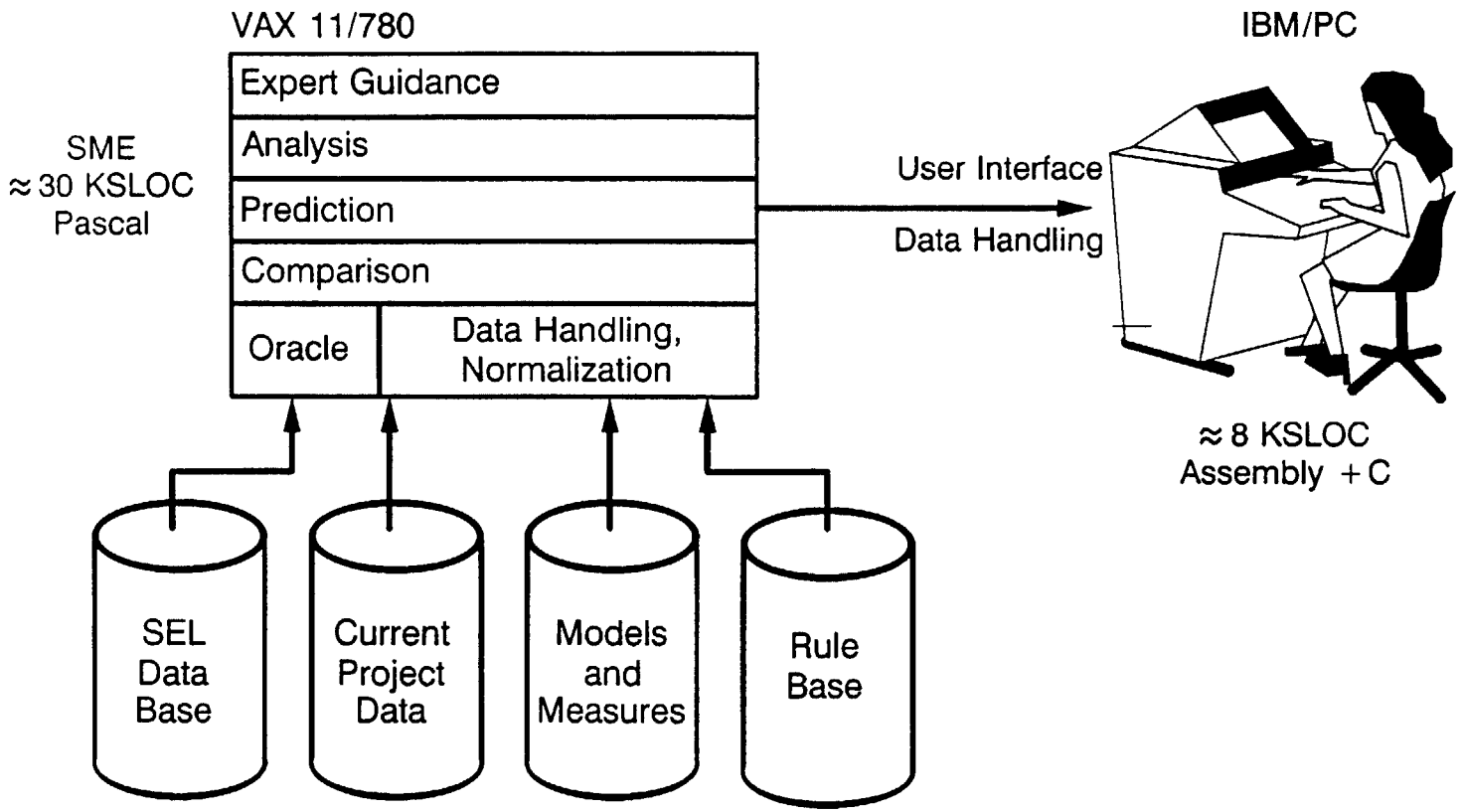
Software development rules have been developed based on previous research and the knowledge of experienced software managers

*Ramsey/Basili: 1986
Valett/Raskin: 1986
Ongoing

J. Valett
NASA/GSFC

C218.007

SME ARCHITECTURE

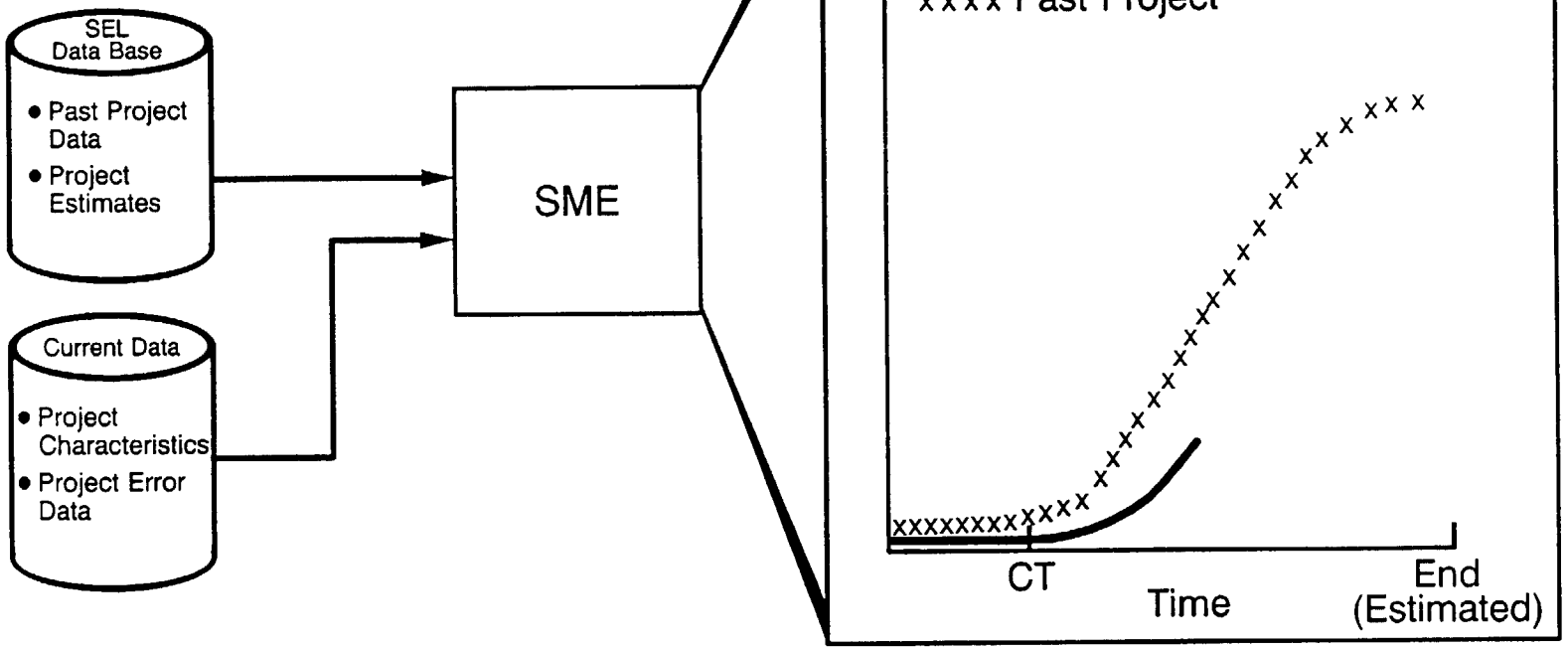


SME utilizes the SEL data base, models, and the rule base to provide managers with comparison, analysis, prediction and expert guidance

C218.008

EXAMPLE OUTPUT

How does my project **COMPARE** with one past similar project?

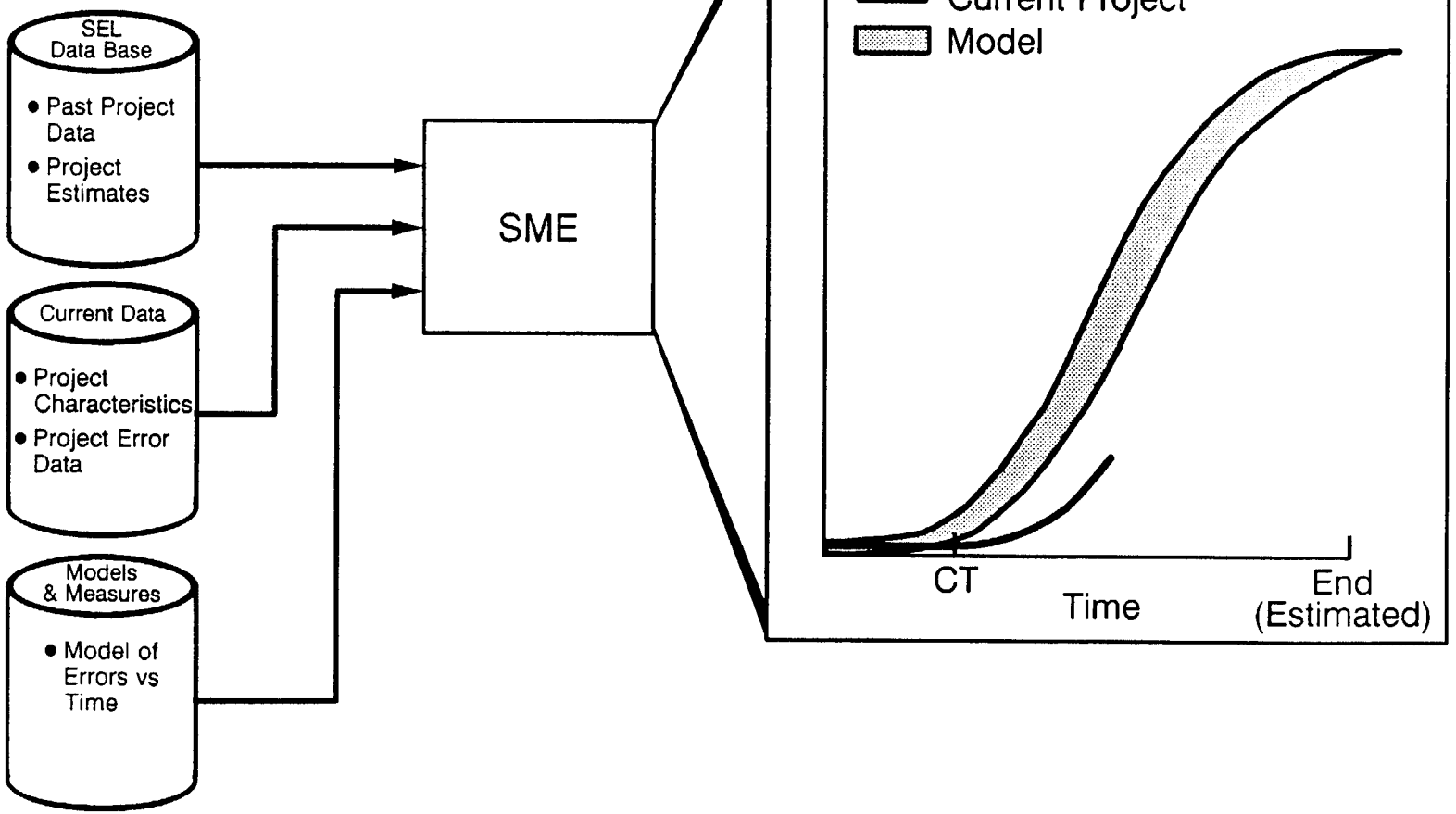


J. Valett
NASA/GSFC
17 of 21

C218.009

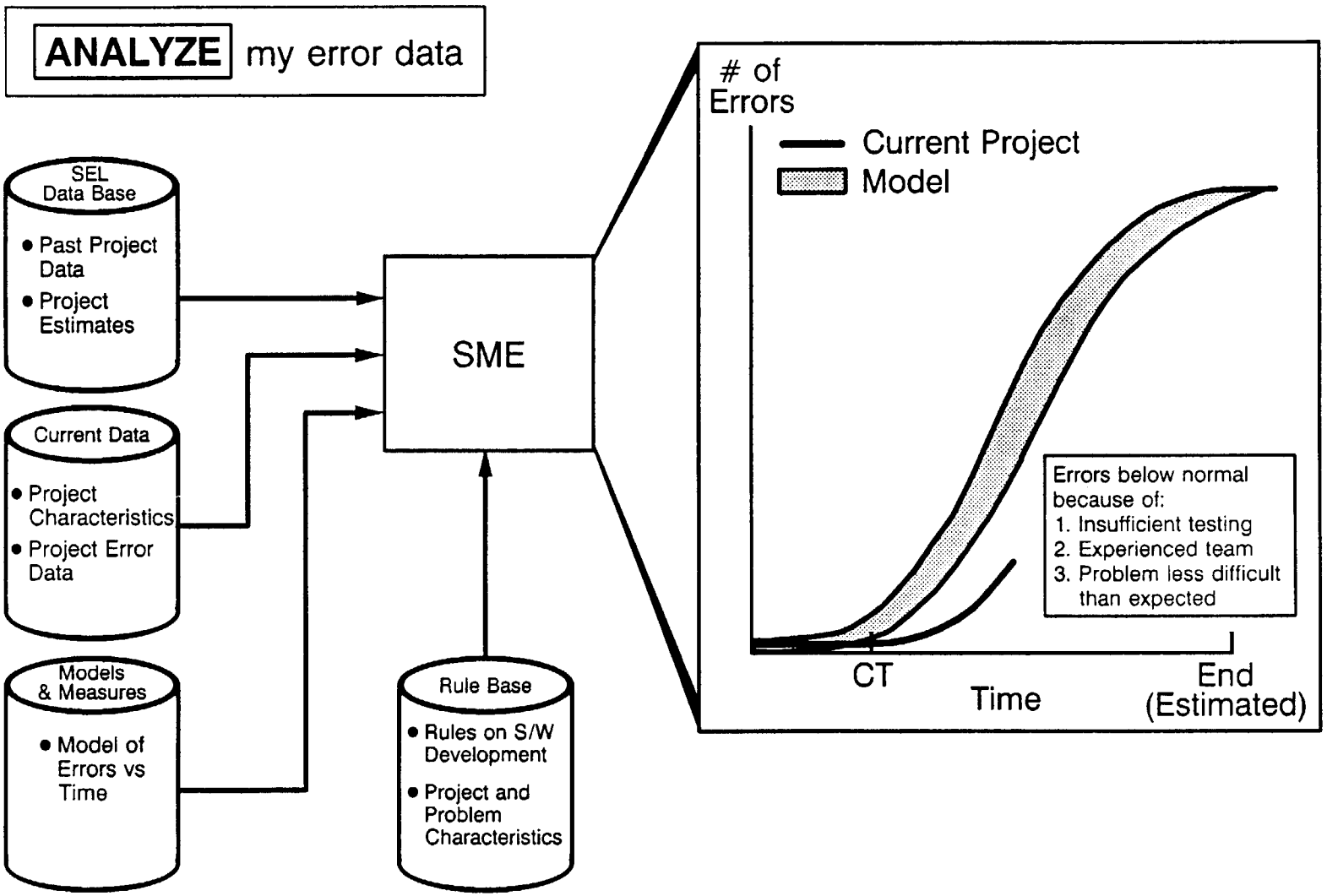
EXAMPLE OUTPUT

How does my project **COMPARE** with the typical project in the environment?

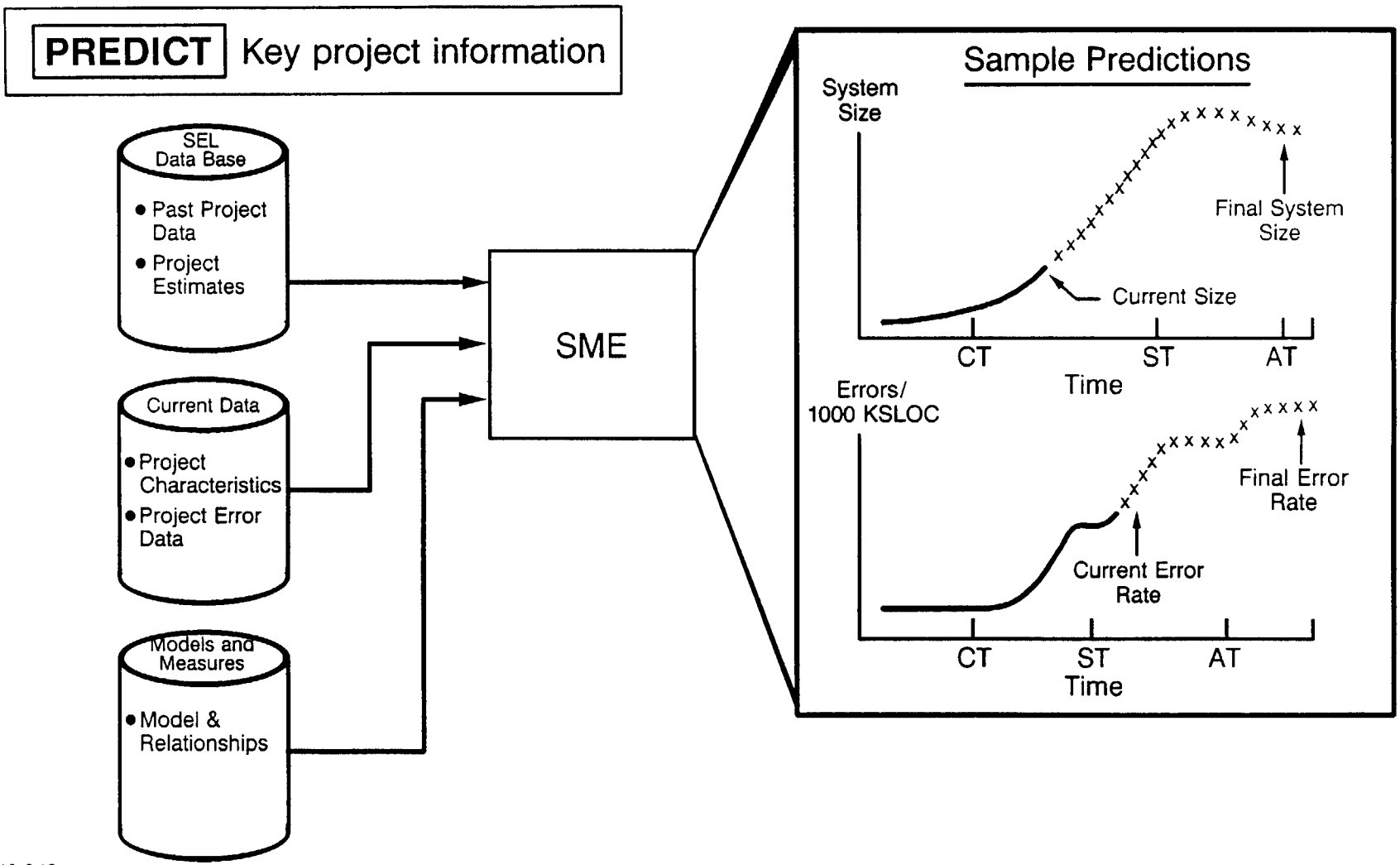


C218.010

EXAMPLE OUTPUT



EXAMPLE OUTPUT



C218.012

FINAL POINTS

1. Software Measurement Data
   - Used as feedback too infrequently
   - Easy to get data can be very useful as a management aid
2. The Maturing of SME
   - SME concept is to continually evolve
   - Dynamically develop models and relationships
   - Learn "new" rules
   - Improving/refining knowledge of environment
3. Focus
   - Assumes waterfall life cycle model
   - Assumes homogeneity of problems/environment
     BUT...
     SME will evolve
4. Currently
   - Not available for general use/distribution, yet
   - Evaluation of applicability/accuracy underway

PANEL #2

SOFTWARE MODELS

R. Tausworthe, Jet Propulsion Labortary
T. Henson, IBM
W. Cheadle, Martin Marietta

D 88-25

A COMMUNICATION CHANNEL MODEL OF THE SOFTWARE PROCESS

Robert C. Tausworthe

October 15, 1988

National Aeronautics and Space Administration

JPL

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Abstract

This publication reports beginning research into a noisy communication channel analogy of software development process productivity, in order to establish quantifiable behavior and theoretical bounds. The analogy leads to a fundamental mathematical relationship between human productivity and the amount of information supplied by the developers, the capacity of the human channel for processing and transmitting information, the software product yield (object size), the work effort, requirements efficiency, tool and process efficiency, and programming environment advantage. The publication also derives an upper bound to productivity that shows that software reuse is the only means that can lead to unbounded productivity growth; practical considerations of size and cost of reusable components may reduce this to a finite bound.

1 INTRODUCTION

As Boehm [1] notes in a recent article, the computer software industry for years has been accused of inferior productivity in comparison to its hardware counterpart, whose productivity continues to increase at an intense rate. Despite advances in languages, development environments, work stations, methodologies, and tools, software projects seem to continue to grind out production-engineered code at about the same old 8 to 15 delivered lines of source code per staff-day. Yet, as Boehm also points out, if software is judged using the same criteria as hardware, its productivity looks pretty good. One can produce a million copies of a developed software product as inexpensively as a million copies of a computer hardware product. The area in which productivity has been slow to increase is the development and sustaining phases of the software life cycle.

Profit-making organizations may amortize their software development and sustaining costs over large customer markets, so that low development productivity is mitigated by larger and larger markets. But government agencies, their contractors, and non-profit organizations must rely on increases in productivity to avoid costs and improve quality. Development and sustaining costs are not often recovered by duplicating the product many, many times.

Software development and sustaining productivity has been the subject of many articles to date. It is also the focus of this publication, which is, in a sense, a mathematical proof of Brooks' [2] assertion that "there is no silver bullet." The avenues for productivity improvement have been adequately summarized by Boehm [1] as

1. Get the best from people.
2. Make the process more efficient.
3. Eliminate steps where possible.
4. Stop reinventing the wheel.
5. Build simpler products.
6. Reuse components.

All of Boehm's steps above, except the first, are human-information-input reductive. Software tools, aids, support environments, workstations, office automation, automated documentation, automated programming, front-end aids, knowledge-based assistants, information hiding, modern programming practices, life-cycle models, common libraries, application generators, next-generation languages, *etc.* all save labor by supplying or modifying information at a faster rate or more reliably than can be done by humans.

Software is information for computers that is made from information supplied by people. Some of the human input information may be new, and some

may be reused, perhaps altered for the new application. Some of the output information product is thus new, and some may derive from legacy, perhaps altered for the function intended. It is therefore intuitive to think of productivity in terms of the amount of information appearing in the output product relative to the effort required from humans to supply the needed information relating to that product. We shall more precisely define productivity using this concept a little later; for the present, let us merely acknowledge that software production capacity increases when the effort required from humans in supplying the information needed to construct a given product is reduced.

It is reasonable, then, to put information and communication theory to work on the theoretical capacity of productivity. In 1949, Claude Shannon [3] proved that communications channels have theoretical information transmission rate limits that are influenced by their channel configurations, signal-to-noise ratios, and bandwidths[1]. Humans and computers developing software are communications devices and channels, and therefore subject to Shannon's law. *Humans are capable of transmitting information only at a rate below their capacity limit* [4]. The channel may transmit more data volume than the actual number of *information bits* due to redundancy and encoding; however, the information rate of bits emanating from the output (*i.e.*, the output entropy) may not exceed the rate that information bits are input (the input entropy). In the parlance of information theory and thermodynamics, there can be no "Maxwell's demons" in the channel.

When building an information product, part of the input information needed is in the form of "black box" specifications of functional and performance requirements. Some of this is new, supplied by humans, and some of it is old, retrieved from other existing sources. But which portions of the old information are to be reused, and how they are to be located, extracted, modified, and integrated with the new information comprises more new information that also must be (largely) supplied by humans.

Once a new or modified software product has been developed, both it and its components are candidates for reuse in forthcoming software products. Thus, the repertoire of reusable objects may grow without bound as the industry wends its way into the future. Reusable objects may be envisioned as new functions appended to an extensible implementation language that may be used in the next project. The conceptual minimum information required at the human input interface is merely that required to select the language features to be used and to integrate them properly into the operating product(s). In the ideal, we may look to automated and knowledge-based tools to supply the other necessary searching, manipulative, transformational, and inferential information associated with matching function-to-language-feature correspondences, integration and construction of the product, and validation.

The question arises, then, can the information content of the output products

[1] The most popular form [3] of Shannon's law is $C_0 = B \log_2(1 + S/N)$.

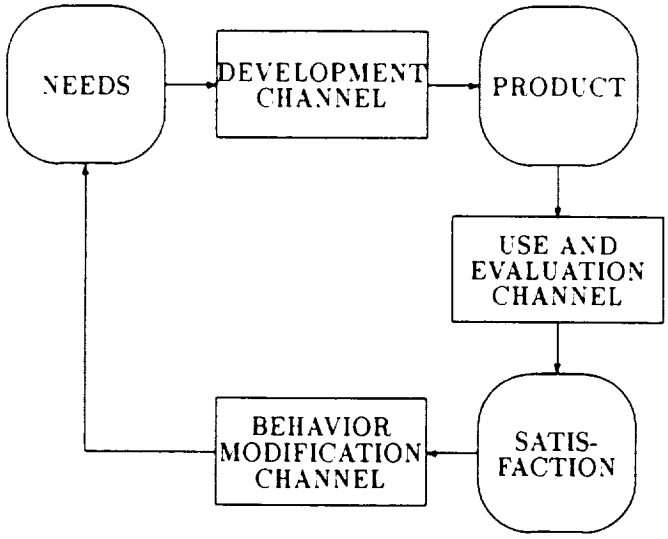


Figure 1: An abstract product life cycle process.

in such an ideal software environment continue to grow at a faster rate than the input rate, or is productivity growth limited by some form of "Shannon limit?" If so, what are the factors which control that limit? This publication develops a framework for answering these questions and characterizing the solutions.

2 THE COMMUNICATIONS ANALOGY

The discussion above characterizes the software development process as one in which, as in Figure 1, various kinds of information are supplied by humans toward implementing a product whose form is also information: documents, programs, parametric data, databases, and test data. Software development is thus an Information-Input/Information-Output (I^3O) process. In like fashion, the use and evaluation of software products are also I^3O processes. Even the behavior modification that shapes needs based on the level of satisfaction derived from use and evaluation of the products is, to some extent, an I^3O process.

An I^3O process may thus be portrayed, for purposes here, as a noisy communication channel with the following traits:

1. *Transformational.* Output information (*i.e.*, the product) exists in a different form than provided in the input (*i.e.*, requirements).

2. *Distortive.* Some input requirements may be implemented differently than

intended.

3. *Erasive.* Some of the input requirements may not have been implemented.

4. *Spurious.* Some features implemented may not have been specified in the input requirements.

5. *Random delay.* The transport time from requirements to product is a variable time, only partially predictable.

6. *Random cost to use.* The cost in dollars and effort needed to transform requirements into products is only partly predictable. The cost of products is the cost of operating the channel.

7. *Non-stationary.* The uncertainty aspects of the channel vary with time.

As is true of other communications systems, the channels themselves must be constructed before they can be used, at a certain cost. I^3O channels consist of people and machines working in randomly connected orchestration. Moreover, the I^3O channels that are used to construct products are themselves the products of other I^3O channels. Thus, if carried too far, the analogy becomes more intricately interconnected, complex, and difficult to analyze, but perhaps more true to life.

Software problems restated in terms of I^3O channels are:

- channel costs are too high.
- throughput delay is too long.
- input/output correlation is too low and difficult to validate.
- input and output are not entirely quantifiable, consistent, nor tangible.
- cost, delay, and throughput are not entirely predictable nor controllable.

More microscopically, an overall communications channel may be viewed as an interconnected network of noisy components and sub-channels. In analogy, high-level software problems decompose into smaller interrelated contributory problems, deriving from many sources. During the conceptualization, requirements capture, and alignment processes of the product cycle, distortion and noise (faults) derive from unknown or unrecognized needs, unexpressed needs, wrongly expressed needs, conflicting needs, non-stationary needs, and inability to quantify and articulate needs. During the implementation and alteration stages, noise comes from misunderstood or ambiguous requirements, conflicting views of utility, inability to simulate a product in entirety, inadvertent omission, conflicting requirements, and unfeasible requirements. During the testing and validation stage, difficulties arise in the combinatorial impracticality of certainty, in the need for an operational environment in some actual or simulated

form, in the need for the product in a simulated or completed, mature form, and in the need for definitive acceptance criteria. Ultimately, the evaluation and enlightenment processes require products and operational environments in completed or simulated form, and are exposed to imprecise, subjective, intangible satisfaction criteria.

Typical considerations which relate to, contribute, or cause these problems are the complexity of the I^3O channels and the products they produce, the stochastic behavior of people, and rapidly changing hardware and software technology. Moreover, our understanding of the software process is still in its evolutionary stage: Tools, environments, and systems are only moderately sophisticated. Methods, models, and theoretical bases for development and product analyses are sparse and largely invalidated. Preparation of products for legacy has often not been properly consummated during development. The reuse of inheritance has been difficult, even when legacy goals are adequately set and fulfilled. Automated knowledge bases for software engineering and applications domains are in their infancy. The transmission medium (*i.e.*, human language) lacks precision in many contexts. And, finally, the skill base of software personnel has not yet been adequately oriented to a disciplined, standardized, industrial-strength engineering approach.

Feedback is commonly used in electronics to stabilize performance. However, the high costs and long delays in I^3O channel usage tend to inhibit firm, immediate feedback for risk of fomenting an unstable situation and incurring yet higher implementation costs and longer delays.

The communication system approach to improvement of channel performance, however, is simple and straightforward:

1. Measure and characterize the channel and its parameters.
2. Expect transmission to be distorted, noisy, and delayed, and provide appropriate compensation.
3. Design the information throughput rate to be within channel capacity (as, e.g., Shannon's limit, or other formula applying to the particular channel[2]).
4. Remove redundancy in the source information before transmission.
5. Make the transmitted information be resilient to channel disturbances by using effective encoding and decoding techniques.
6. Transmit information through the channel with as great a signal force as possible.
7. Take steps to reduce disturbances within the communications channel.

[2]Software production capacity in the absence of fault generation and correction is given by Eq. 25.

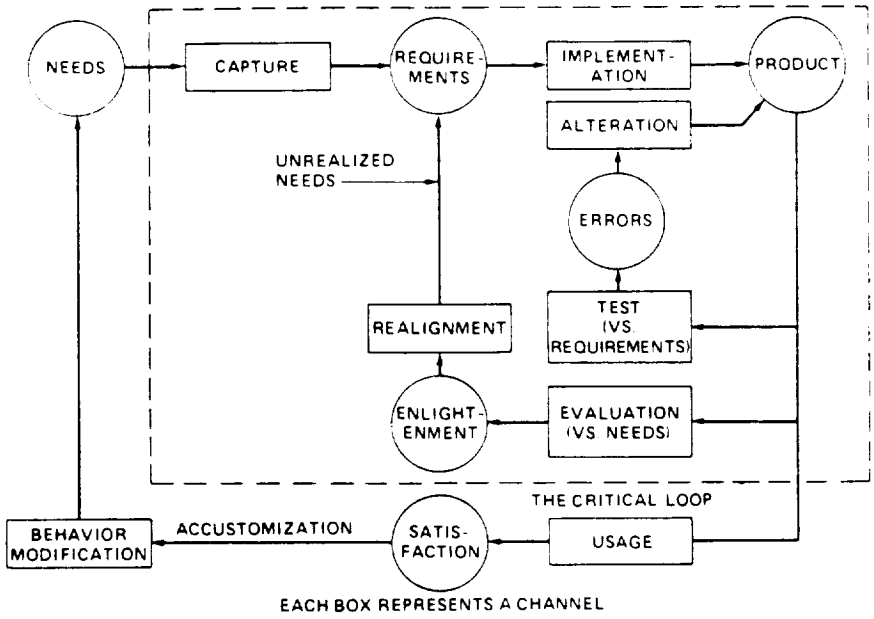


Figure 2: The I^3O life-cycle channel model.

8. Use feedback to correct errors.

The goal of this publication, then, is to characterize and quantify software production in analogy with communications theory, and thereby in terms of measurable, causal, and controllable factors.

3 THE SOFTWARE CHANNELS

A basic idealized production configuration was depicted in Figure 1, where needs are faithfully projected in the form of information through the development channel to yield information products, which are then used, evaluated, and may lead to a certain level of satisfaction. Use and accustomization beget behavior modification, which, in turn, elevates the original set of needs toward higher levels of automation. Not present in this ideal are the intrinsic distortions, faults, and other flaws that produce less-than-ideal products, incomplete levels of satisfaction, and, perhaps, unfortunate modifications of behavior that limit the tendency toward higher automation.

A refinement of this concept is shown in Figure 2, where the processes associated with channel imperfections are displayed more prominently. Needs are projected through a capture channel to produce a requirements specification; requirements are transmitted via an implementation channel into the product set; the product set is put through a testing channel to reveal (some of the) errors; errors are fed into the alteration channel, which (partially) corrects the product set; evaluation of the product set against stated requirements often reveals shortcomings, leading to an enlightened state; and enlightenment guides the process of requirements realignment. Usage of the product set, as earlier, produces a level of satisfaction (not necessarily complete), which alters the state of need through behavior modification.

Each of the information transmission channels and information sets can be further dissected and detailed for better understanding of the transformation processes and better accuracy in modeling the software phenomena.

The critical, and perhaps less philosophical, portion of the refined software channel analogy is shown inside the dashed lines of Figure 2. This portion comprises the software development and sustaining segments of the life cycle. Note that the analogy can be made to simulate information transmission aspects of the "ordinary waterfall" life cycle, incremental development, rapid prototyping, evolutionary enhancement, and "spiral" life cycle paradigms merely by suitable definitions of channel characteristics. In the next section, the software channel analogy is used to develop a refinery model of software productivity, to which information and communication theory are applied to derive statistical limitations on human capacity to produce larger and larger software systems.

4 THE IMPLEMENTATION CHANNEL

The assumed software implementation components are illustrated in Figure 3. Five forms of information input by humans are identified: requirements (function, performance, and constraints), transformational (design and coding), combinational (integration), corroborative (validation and verification), and management (status and control). Each of these potentially contains imperfections in the form of accidents (inadvertent, random faults) and distortions (deliberate, non-random faults). Together, these latter two constitute a sixth type of information input by humans that we shall collectively refer to as *noise*. Also shown is the set of products resulting from the inputs.

Generation and application of the above input information to the software implementation channel is assumed to constitute the entire expenditure of human effort. Information generated by humans is mental, verbal, and documentation, and only the last of these is amenable to measurement. We must, therefore, hypothesize that the capture of information in memoranda, documents, code and comments, parametric and test data, *etc.*, is representative of and correlates significantly with the total outlay of effort.

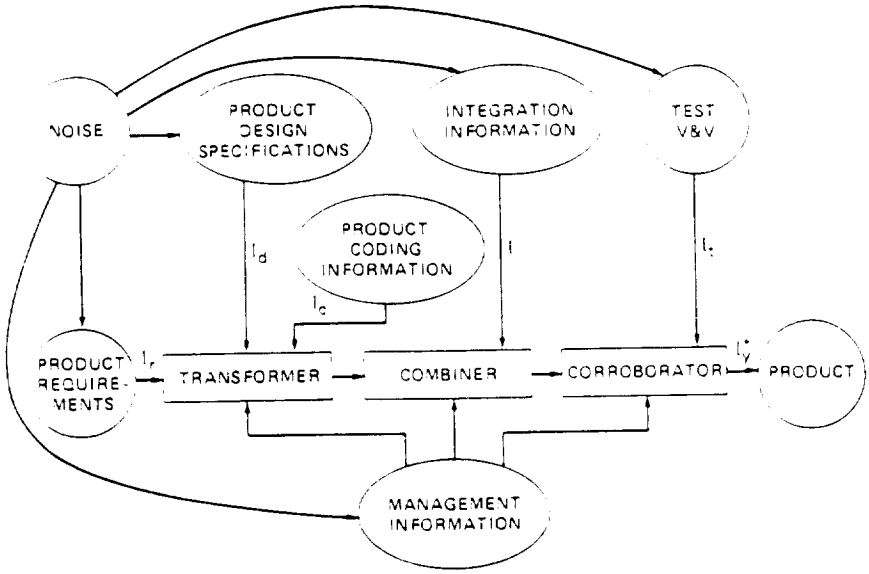


Figure 3: The software production refinery.

Output products are viewed as condensations, transformations, and refinements of the information that came into the environment; hence, we refer to the implementation process as the **Software Refinery**. Productivity improvement in the refinery is tantamount to reducing the amount of human-supplied input information required for a given output product set.

Effort-intensive input information requirements will be minimized by eliminating redundancy and by reusing existing information whenever feasible. For example, if a system has a requirement for a word processor of a known type, then the single expression "Wordstar[3] 4.0" could be used to convey unambiguously all the characteristics that the cited word processor possesses. Moreover, if there were only $1024 = 2^{10}$ word processors in the world, only 10 bits would be needed to distinguish Wordstar among its competitors. Only exceptional and incremental information would be then be needed to specify a slightly different capability desired. Additionally, since Wordstar already exists, further information relating to design, implementation, and testing is not required, except where it relates to the integration of that package into the system being built.

Also, when documents must be developed to contain previously generated information (*i.e.*, "boilerplate"), the only information conceptually required from

[3]Wordstar is a registered trademark of MicroPro, Inc.

the human is where to find the boilerplate, how much of it to use, where to put it, and any necessary alterations.

For the remainder of this publication, we shall focus on that information leading to the program (set), or *product yield*. Therefore, effort and information used to produce documents is limited to that which is yield related. These include requirements documents, design specifications, project plans and status reports, test plans and procedures, and the like; preplanning, applications, operations, and maintenance documents are excluded at this time. We have hypothesized that the information content of these entities correlates strongly with the total project information. By measuring the information contents of software project documents and output yields, then, quantitative relationships among input information and output yield may be established.

Transformational and corroborative information input needs are potentially reduced by reusing elements of previous designs and code whenever feasible. In the ideal, fully automated case, this reduction could be almost complete: automated catalogs of solved problems would be searched using knowledge bases having extensive application domain-dependent inference and design rules that match functional and performance requirements with known solutions and designs, designs with working code, *etc.* In the ideal automated software refinery, the amount of input noise, and thus the need for corroborative information, could also be drastically reduced. The ideal software refinery is shown in Figure 4.

Although much of the integrative information would also conceptually be supplied by automation, some will nevertheless still be required from humans to relate interdependency among functional features, data flows, and orders of precedence.

We model the software production refinery in the form of an extensible language. That is, the human information input $\mathcal{I}$ is used to develop the output yield $\mathcal{Y}$ from new information and from instructions to reuse existing information and previously developed parts that operate within given time and data precedence constraints.

The distinguished components of the input $\mathcal{I}$ are (Figure 3)

$$\mathcal{I} = \mathcal{I}_r \cup \mathcal{I}_d \cup \mathcal{I}_c \cup \mathcal{I}_i \cup \mathcal{I}_t \cup \mathcal{I}_m \tag{1}$$

These terms represent, respectively, *r*equirements, *d*esign, *c*ode production, *i*ntegration, *t*est (including validation and verification), and *m*anagement information sets. Each of the input sets potentially contains faulty information, or noise.

In particular, we shall assume that the requirements term, $\mathcal{I}_r$, can be isolated to contain the functional, performance, and algorithmic specifications and constraints, so that, in concept, a fully automated programming environment could produce the output yield in the current refinery without further information.

We define the *inherent product specification*, $\mathcal{I}^*$, as the least practical information required to specify the output yield uniquely. It is the mapping of the

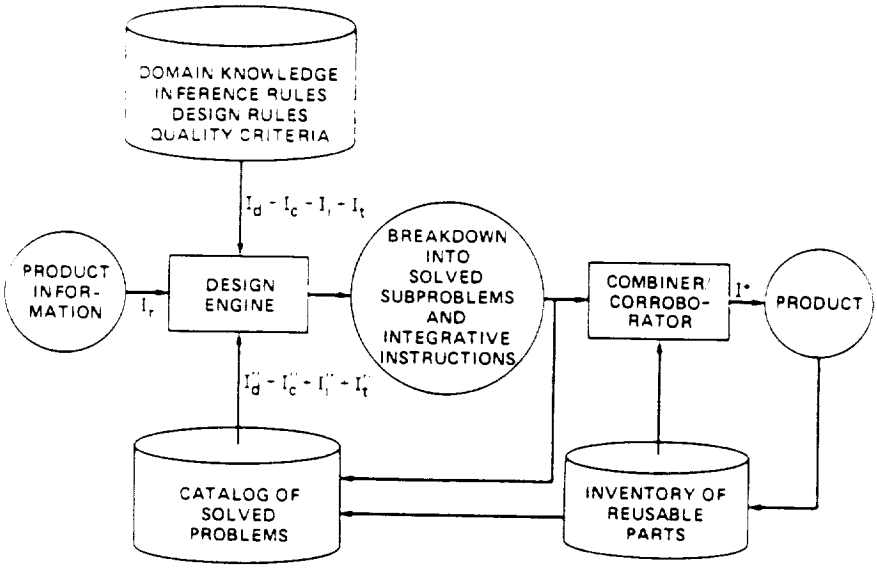


Figure 4: The ideal software refinery configuration.

input information through the production transformation

$$\pi(\mathcal{I}) = \mathcal{I}^* \tag{2}$$

Conversely, that subset of the input, denoted $\hat{\mathcal{I}}$, that traces to the as-built product is defined by the inverse production transform,

$$\pi^{-1}(\mathcal{I}^*) = \hat{\mathcal{I}} \tag{3}$$

Note that this traceability may not necessarily be direct: Constraints, performance requirements, and design goals in $\mathcal{I}$ certainly influence the resulting $\mathcal{I}^*$; but it may be difficult indeed to correspond any tokens of the output product with tokens of the input information. Therefore, $\hat{\mathcal{I}}$ should be regarded as that (amended) form of $\mathcal{I}$ that got built.

The sets of fulfilled and unfulfilled requirements are described by

$$\mathcal{I}_f = \mathcal{I}_r \cap \hat{\mathcal{I}} \tag{4}$$

and

$$\mathcal{I}_e = \mathcal{I}_r - \mathcal{I}_f \tag{5}$$

respectively. That is, $\mathcal{I}_f$ is that portion of $\mathcal{I}_r$ that got implemented, and I_e is the remainder of $\mathcal{I}_r$.

The executable program, or *apparent yield* $\mathcal{Y}$ will include the inherent product specification, $\mathcal{I}^*$, as well as the $\mathcal{I}_i^*$ of each of the n modules in the refinery invoked by $\mathcal{I}^*$, as transformed by the compiler and linker into a functioning unit. $\mathcal{Y}$ will normally be sensitive to compiler and linker characteristics, such as type and degree of code optimization, extent of program and data segmentation, *etc.* Thus, we define the *inherent functional yield*, $\mathcal{Y}^*$, as the join of inherent product specifications over all components comprising the final product.

$$\mathcal{Y}^* = \bigcup_{i=0}^{n} \mathcal{I}_i^* \tag{6}$$

in which $\mathcal{I}_0^* = \mathcal{I}^*$.

We denote the sizes of these sets by

$$I_k = |\mathcal{I}_k| \quad \text{for } k = r, d, c, i, t, m \tag{7}$$

$$I = |\mathcal{I}| \leq I_r + I_d + I_c + I_i + I_t + I_m \tag{8}$$

$$I^* = I_0^* = |\mathcal{I}^*| \tag{9}$$

$$I_i^* = |\mathcal{I}_i^*| \quad \text{for } i = 1, \ldots, n \tag{10}$$

$$Y = |\mathcal{Y}| \tag{11}$$

$$Y^* = \sum_{i=0}^{n} I_i^* \tag{12}$$

Naturally, $I^* \leq I_r$ by Shannon's law, and *a fortiori* $I^* \leq I$. Also, $I^* \leq Y^*$ because $\mathcal{I}^* \subseteq \mathcal{Y}^*$.

Input information is perhaps most meaningfully measured in terms of the *chunks* [4] that humans treat as units of information in memory and recall. However, the mechanism for chunking is not yet well enough understood (at least, by the author) to be able to compute an input information chunk measure. Rather, the first-order entropy [3] based on word and symbol, or *token*, counts and vocabulary usage will be used:

$$H_k = -\sum_{i=1}^{R_k} p_{k,i} \log_2 p_{k,i} \qquad \text{for k = r, d, c, i, t, m.} \tag{13}$$

$$I_k = N_k H_k \tag{14}$$

Here, R_k is the size of the *R*epertoire, or vocabulary, of words and symbols used in $\mathcal{I}_k$, $p_{k,i}$ is the relative frequency in usage of the i-th word or symbol in that repertoire, and N_k is the total number of words and symbols used. Since words and symbols represent first-order chunking by humans, the information first-order entropy measures should correlate strongly with information measures based on chunking. Evaluation of higher-order entropy (phrases, syntactic forms, *etc.*) may be appropriate for study at a later date.

Segments of documents that are included from other sources should not be counted this way, because the apparent information content would be higher than that actually supplied by humans (this time) for its reuse. If such portions can be handled separately, the true human input involvement can more accurately be approximated.

We similarly characterize the inherent input content $\mathcal{I}^*$ and output yield $\mathcal{Y}^*$ in terms of the features of the extensible language. Let R be the number of unique operators and operands that already exist in the current refinery language repertoire, or vocabulary. This number will include both the basic set of built-in functions, as well as every function that has so far been made available to the refinery for reuse (every new function produced is a candidate for reuse, if applicable and feasible). Next, let n denote the number of unique refinery operators of this repertoire actually required for implementing the current application. Then, let d signify the actual number of unique input/output data operands appearing in $\mathcal{I}^*$, and let N be the total number of operators and data operands appearing in $\mathcal{I}^*$. Finally, let $\overline{Y}^*_n$ represent the average inherent yield of the n refinery operators invoked by $\mathcal{I}^*$.

The inherent product information $\mathcal{I}^*$ is just sufficient to specify the product yield; in this, it is a translation of $\mathcal{I}_r$ into specific refinery terms. It specifies the needed functions of the repertoire, the inputs and outputs of each, and the integration of these elements into an appropriate sequence of instructions. We note, then, that $\mathcal{I}^*$ is refinery-dependent, because it depends upon the richness of the repertoire at the time of use. To a first-order approximation, $\mathcal{I}^*$ will be equivalent[4] to N instances of $n + d$ unique operator/operand types arranged in proper order. The minimum average number of bits needed to specify any one of the R operators of the current refinery or d data elements of the current operand vocabulary is the first-order entropy H^* of the refinery and data repertoire. Thus, in analogy with Eq. 14,

$$I^* = NH^* = -N \sum_{i=1}^{R+d} p_i \log_2 p_i \tag{15}$$

$$\leq N \log_2(R + d) \tag{16}$$

However, since usage statistics of the refinery and ensemble of applications are unknown at this time, the measure above can only be approximated. For practicality and consistency across languages, the size of the inherent product specification will hereafter in this work be approximated[5] by its upper bound above.

[4]One may need to normalize I^* across semantically equivalent syntactic constructions of the refinery language. For example, the C language form "`x = x + 1`" contains 5 tokens, whereas the form "`x++`" contains only 2. The information content of the two is the same.

[5]Since I^* only appears in the productivity equation in ratio with Y^*, defined in Eq. 18, which is also evaluated in the same way, error due to this approximation will normally be of second order importance.

also known as the Halstead program volume [5],

$$I^* = N \log_2(R + d) \tag{17}$$

Note that language processors, for practicality, generally represent tokens using fixed-bit-length internal representations, rather than by variable, frequency-of-use-derived (entropy based) ones. This practice also requires the use of at least $\log_2(R + d)$ bits per token.

Finally, we express the size of the inherent functional yield as

$$Y^* = I^* + n\overline{Y}_n^* \tag{18}$$

The software refinery model thus provides absolute relationships among the current refinery vocabulary size and the average yield of those operator modules in the refinery that were used. Note that I^*, Y^*, n, and $\overline{Y}_n^*$ can all be determined as measurable properties of the software refinery and the current application program. The reuse portion of the product yield, $Y^* - I^*$, should be measured in the refinery language that would be used to reimplement it, regardless of the language used originally to implement it.

5 THE PRODUCTIVITY EQUATION

Let W denote the total *work effort* (measured in work-months) required to develop an output information product yield $\mathcal{Y}$ from a given information input set $\mathcal{I}$ supplied by humans. *Productivity* is defined here as the inherent functional yield per unit work, in total bits per work-month,

$$P = \frac{Y^*}{W} \tag{19}$$

The use of the inherent functional yield, Y^*, in this definition, rather than the actual apparent yield, Y, which also includes data yield and compiler quirks, is quite arbitrary, but conforms to a practice analagous to counting "executable lines of code," as opposed to "total lines of code." Although Y may perhaps be easier to measure than Y^*, it is, nevertheless, an inadequate indicator of productivity because of its compiler dependence: a better compiler would seem to lower productivity[6].

The average rate at which a given population generates information of a specified type is their *mean work capacity*, C, in bits per work month,

$$C = \frac{I}{W_0} \tag{20}$$

[6]This fact was pointed out to the author by Robert D. Tausworthe of Hewlett-Packard, Inc.

where W_0 is that amount of work required to generate the information $\mathcal{I}$ in an ideal environment where locating existing information, capturing new ideas, and preparing these for use are immediate (*i.e.*, W_0 is measured as the actual work effort minus the location, capture, and preparation effort). C conceptually, then, is a function of problem complexity, human intellect, experience, skill, motivation, work conditions, staff interaction, and emotional and psychological factors.

We know from experience that human capacity has a limit, so we define the *potential information capacity*, C_0, as the ideal value of C that could be achieved if the workers were to be relieved of adverse problem, environment, and human factor encumbrances, and were working at a maximum reliable pace. The unitless ratio

$$\mu = \frac{C}{C_0} \leq 1 \tag{21}$$

then represents a *mental acuity factor*. Since labor wasted in capture and location of information, *etc.*, has been eliminated from μ, it is only independent on environment and tools to the extent that these stimulate individual work capacity. We may note that μ will tend to be greater when $\mathcal{I}$ is produced well within the skill, experience, and understanding of the staff, at a motivated pace of work, and in a smoothly operating and happy organization. However, μ will tend to decrease with other attributes, such as application complexity [1] and staff size [6]. Much of the behavior of μ has been calibrated in various software cost models, where a variation of 500:1 has been noted as necessary to span the range of contributory factors. Consequently, the value of μ for some projects may be on the order of 10^{-3}.

Next, we define *requirements efficiency*, ρ, as the unitless ratio of inherent product specification and requirements information measures,

$$\rho = \frac{I^*}{I_r} \leq 1 \tag{22}$$

This ratio indicates the level of superfluity between information specifying the as-built product and that contained in requirements information. It is partially a natural characteristic of the requirements and refinery languages being used, but also will depend considerably on the style of the individual(s) writing the requirements, the complexity of the problem, the extent to which fulfilled requirements lead to measurable product specifications, the extent to which stated requirements are fulfilled, the amount and distinguishability of new and reused requirements information, and other factors. Measurements of ρ are needed to calibrate the effects of these factors, and to establish norms for its use as a requirements efficiency indicator. A ball-park figure for ρ based on a few document-to-code size estimates is about 0.1.

The ratio of requirements information to total input information reflects the relative degree to which design, coding, test, and management information are required from humans for a given problem. The ratio of W_0 to W is the effort

efficiency in location, capture, and preparation of information. Together, these ratios express the efficiencies of methods, tools, and aids relative to an ideal environment. Labor-saving methods, tools, and aids are those that tend to reduce the amount of *effort* required to generate, capture, or prepare a given amount of information. Examples are word processors, design languages, automated graphics, and data dictionaries. Information-reductive methods, tools, and aids are those that tend to reduce the *amount of information* that is required to be generated by humans. Examples here are symbolic notation, automated design assistants, and test case generators.

We combine these two effects into the *tool factor*, τ, defined as the unitless ratio

$$\tau = \left(\frac{I_r}{I}\right)\left(\frac{W_0}{W}\right) \leq 1 \tag{23}$$

This coefficient reveals the amount of human information, and thus labor, that potentially can be eliminated by methodology, automation, and practice. It provides a simple means by which the effectiveness of solution methods, tools, and engineering processes can be quantified by actual measurements. Note that τ is very likely to be influenced by the *amount* of information that must be processed; the greater I is, the greater the difficulty of the human task in coping with it. Thus, we may expect to see the effectiveness of well-designed tools increase as the size and complexity of the project it is applied to increase. A rough estimate of τ from some document page and approximated human effort ratios is about 0.01.

Finally, the refinery *language advantage*, λ, is defined as the unitless ratio of the reused portion of the output functional yield to the minimum product specification:

$$\lambda = \frac{Y^* - I^*}{I^*} = \frac{n\overline{Y}_n^*}{I^*} \tag{24}$$

This coefficient is quantifiable from token and vocabulary counts in the current refinery model. It represents the information gain factor due to reuse, and signifies how large a product yield can be generated from a minimum product specification in a given refinery environment. Because it is a unitless ratio, λ should be less dependent on a particular refinery than are I^* and Y^* individually, since common tendencies tend to cancel out. A value on the order of about 15 was measured for a group of small C programs using the ANSII standard library functions.

The productivity equation then follows straightforwardly:

$$P = C_0\mu\rho\tau(1+\lambda) \tag{25}$$

$$< C_0(1+\lambda) \tag{26}$$

The productivity formula is intuitive: the smallest sufficient requirements definition, the most effortless implementation, and the most propitious usage of

tools and methodologies yield the highest advantage; reuse of previous products as new available refinery features yields a higher language advantage.

The upper bound above would be replaced by equality under the condition $\mu\rho\tau = 1$, a situation clearly requiring the existence of automatic programming. The bound thus shows that the effectiveness of automated programming environments will be determined by the extent of reuse of components in the refinery. Moreover, *the only route to unlimited productivity growth is through the effective reuse of increasingly larger and larger software components.*

6 LANGUAGE ADVANTAGE TRENDS

It is a remarkable fact that there are statistical laws in natural and computer languages that relate the total number of occurrences of language token types (word types in natural language, and operators and operands in computer languages) to the vocabulary of distinct types used. Laws of this nature were first studied by Zipf [7] in the 1930's in connection with natural languages. Others, notably Halstead [5], Shooman [8], Laemmel [9], Gaffney [10], and Albrecht [11], have extended the study to computer languages and specifications.

The assumption of the method is that the specifications and the programs that embody those specifications are two descriptions of the same thing. Knowledge of one correlates with knowledge about the other. For example, it is reasonable to expect that a statement of basic requirements for a program includes an itemization of its inputs, processing, and outputs, viewed externally. This external statement translates, through the works of Zipf, Halstead, and the other authors cited above, into approximate measures of the output product yield. These measures generally agree within about a factor of 2; hence, we introduce a factor ζ to account for the difference between Zipf's first law and the true refinery model token length characteristic.

Zipf's first law, for example, predicts the approximate token length $\hat{N}$ of $\mathcal{I}^*$ as the value

$$\hat{N} = (n + d)[\gamma + \log(n + d)] \tag{27}$$

where γ is the Euler constant, $\gamma = 0.57721\ldots$. The factor $\zeta = \hat{N}/N$ makes the equation exact, by definition:

$$N = \frac{1}{\zeta}(n + d)[\gamma + \log(n + d)] \tag{28}$$

The token-length correction factor ζ fluctuates from program to program, but ranges approximately between 0.5 and 2.

The refinery language advantage, therefore, is

$$\lambda = \frac{\zeta\, n\, \overline{Y}^*_n}{(n + d)\log_2(R + d)[\gamma + \log(n + d)]} \tag{29}$$

$$< \frac{\zeta \overline{Y}_n^* \log 2}{\log n \log R} \tag{30}$$

$$< \frac{\zeta \overline{Y}_n^* \log 2}{\log^2 n} \tag{31}$$

which, as may be noted, is limited only by average utilized module yield and vocabulary size. As they stand, these expressions are not statistical: $\lambda, \zeta, \overline{Y}_n^*$, and n are determined by the particular program. Averaging λ over an ensemble of programs would yield a statistical bound, however, of the form

$$\overline{\lambda} < \frac{\overline{\zeta}\, \overline{Y}_{\overline{n}}^* \log 2}{\log^2 \overline{n}} \tag{32}$$

for $\overline{\lambda} = E(\lambda)$ and appropriately defined $\overline{\zeta}$ and $\overline{n}$. This statistical form of the bound reveals that, in order for the refinery language advantage (and thus, productivity) to grow without bound, the average yield of refinery modules being used by applications must grow faster than the square of the logarithm of the number of refinery modules being used. That is, it must happen that modules of increasingly higher yields are regularly added to the refinery and regularly used. *A software refinery with a static, non-expanding library imposes a fixed productivity limit on its workers.*

7 FUTURE WORK

The work reported here is a part of the newly-begun NASA Initiative in Software Engineering (NISE), and is coordinated with other NISE investigations, notably the development of a dual life-cycle paradigm (separating, but interrelating management and engineering processes), the development of a dynamic software life-cycle process simulator, behavioral researches into the performance of humans in the software process, and the synthesis of effective supporting methodologies, tools, and aids.

This first publication reveals only a few rudimentary aspects of the software life cycle process, here modeled as productivity channels refining crude information into highly distilled products. The principle results apply only to the implementation channel, or software refinery. The effects of information noise, the stochastic behavior of people, the detailed character of the other individual component channels, and the dynamic behavior of interacting channels remain to be analyzed and validated.

For the implementation channel, near-term work remains to evaluate C_0, μ, ρ, τ, and λ in a static, low-noise context. Insight into C_0 and μ may be sought in human behavioral research journals. Later work may involve experiments in collaboration with academic researchers.

Typical ρ and τ values may be determined by measurement of documents and programs in existing project libraries for which effort statistics are available;

regression with perceived contributory factors would then quantify effects and suggest avenues for productivity improvement. Studies of τ and ρ may be expected to calibrate benefits of selected methodologies and tools.

Still other studies remain to examine the statistical behavior of λ as a function of the refinery size and reuse policy, to determine whether there are natural limits to productivity growth, and thus, to resolve the question posed by the upper bound in Eq. 31 above.

Further research will quantify the behavior of the other component channels of the production life-cycle model, as well as the dynamic interaction of information flows in the model, notably those within the critical loop shown in Figure 2.

8 CONCLUSION

This publication has developed a model of the software implementation process that formulates productivity as a product of tangible, definite, measurable, and meaningful factors. The model characterizes productivity as stemming from five weakly interrelated factors: human information capacity, mental acuity, requirements specificity, methodology and tool efficiency, and refinery language advantage. Each of these factors was shown to have absolute, explicit, and measurable bounds: Human performance is limited by inherent human channel capacity and by the degree of mental acuity that can be achieved toward realizing that capacity. Requirements efficiency is limited by the minimum as-built product specifications and the extent to which requirements specifications can be freed from extraneous, superfluous material. The effectiveness of tools and methodologies is limited to the amount of human (labor) input that can be avoided. And finally, the effectiveness of a programming environment is limited by the average growth in yield of modules in that environment.

These factors serve as absolute standards for comparison purposes: μ reveals how well the staff are meeting their potential; ρ expresses the level of superfluity of requirements; τ quantifies the effectiveness of methodologies, tools, and aids; and λ indicates the power of the refinery. Use of these standards will lead to meaningful tradeoffs and, potentially, to an eventual optimized software life cycle.

References

[1] Boehm, Barry W., "Improving Software Productivity," *Computer*, IEEE Computer Society, Vol. 20, No. 9, 1987, pp. 43-57.

[2] Brooks, Fred P., "No Silver Bullet—Essence and Accidents of Software Engineering," *Proc. IFIP Congress 1986*, North-Holland, 1986, pp. 1069-1076.

[3] Shannon, Claude E., "Communication in the presence of noise," *Proceedings of the I.R.E.*, Vol. 37, 1949, pp. 10-21.

[4] Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capability for Processing Information," *Psychology Review*, March, 1956, pp. 81-97.

[5] Halstead, M. H., *Elements of Software Science*, Elsevier North-Holland, Inc., New York, NY, 1977.

[6] Brooks, Fred P., *The Mythical Man Month*, Addison-Wesley Publishing Co., Reading, MA, 1975.

[7] Zipf, G. K., *The Psychobiology of Language: An Introduction to Dynamic Philology*, Houghton Mifflin, Boston, MA, 1935.

[8] Shooman, M. L., *Software Engineering*, McGraw-Hill Book Co., New York, NY, 1983.

[9] Laemmel, A. E., and Shooman, M. L., "Statistical (Natural) Language Theory and Computer Program Complexity," Polytechnic Institute of New York, Report POLY-EE/EP-76-020, August, 1977.

[10] Gaffney, J. E., "Software Metrics: A Key to Improved Development Management," *Computer Science and Statistics: Proc. of the 13th Symposium on the Interface*, Springer-Verlag, New York, NY, pp. 211-220.

[11] Albrecht, A. J., and Gaffney, J. E., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 6, November 1983, pp. 639-648.

THE VIEWGRAPH MATERIALS

FOR THE

R. TAUSWORTHE PRESENTATION FOLLOW

A COMMUNICATIONS CHANNEL ANALOGY MODEL OF THE SOFTWARE PROCESS

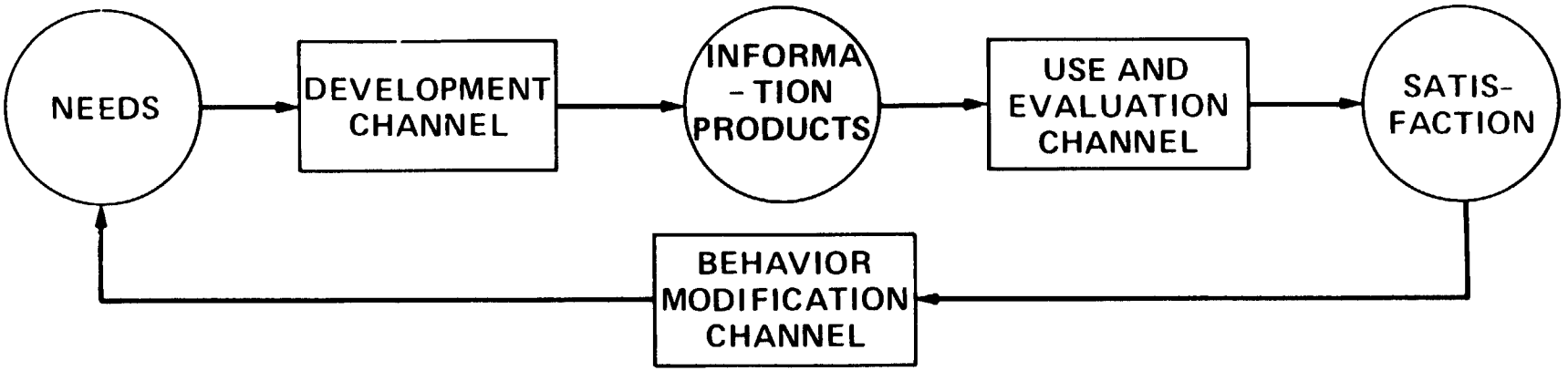
Robert C. Tausworthe

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91009

RCT-1
11-30-88

JPL

THE SOFTWARE PROBLEM MODEL: A NOISY COMMUNICATION CHANNEL



- INFORMATION IS WHAT FLOWS IN THE CHANNELS
- CHANNEL CHARACTERISTICS
  - TRANSFORMATIONAL
  - DISTORTIVE: LOSS IN FIDELITY
  - ERASIVE: LOST FACTORS
  - SPURIOUS: EXTRANEOUS FACTORS INTRODUCED
  - RANDOM DELAY
  - RANDOM COST TO USE
  - NON-STATIONARY CHARACTERISTICS
- CHANNELS, PRODUCTS, AND SATISFACTION HAVE TO BE BUILT
  - THEY ARE THE PRODUCTS OF OTHER SIMILAR CHANNEL MODELS
  - THE OVERALL MODEL IS EXTREMELY INTRICATE AND EVOLUTIONARY
- CHANNELS ARE MADE UP OF PEOPLE, HARDWARE, AND ENVIRONMENTS

JPL

"THOSE UNENLIGHTENED PROGRAMMERS HAVE NEVER HEARD OF THE THEORY OF FUDLIPS. I'LL DO THEM A FAVOR BY COUCHING THEIR PROBLEMS IN FUDLIP NOTATION, AND THEN EVERYTHING WILL BE SOLVED BECAUSE ALL THE THEOREMS ABOUT FUDLIPS CAN BE APPLIED."

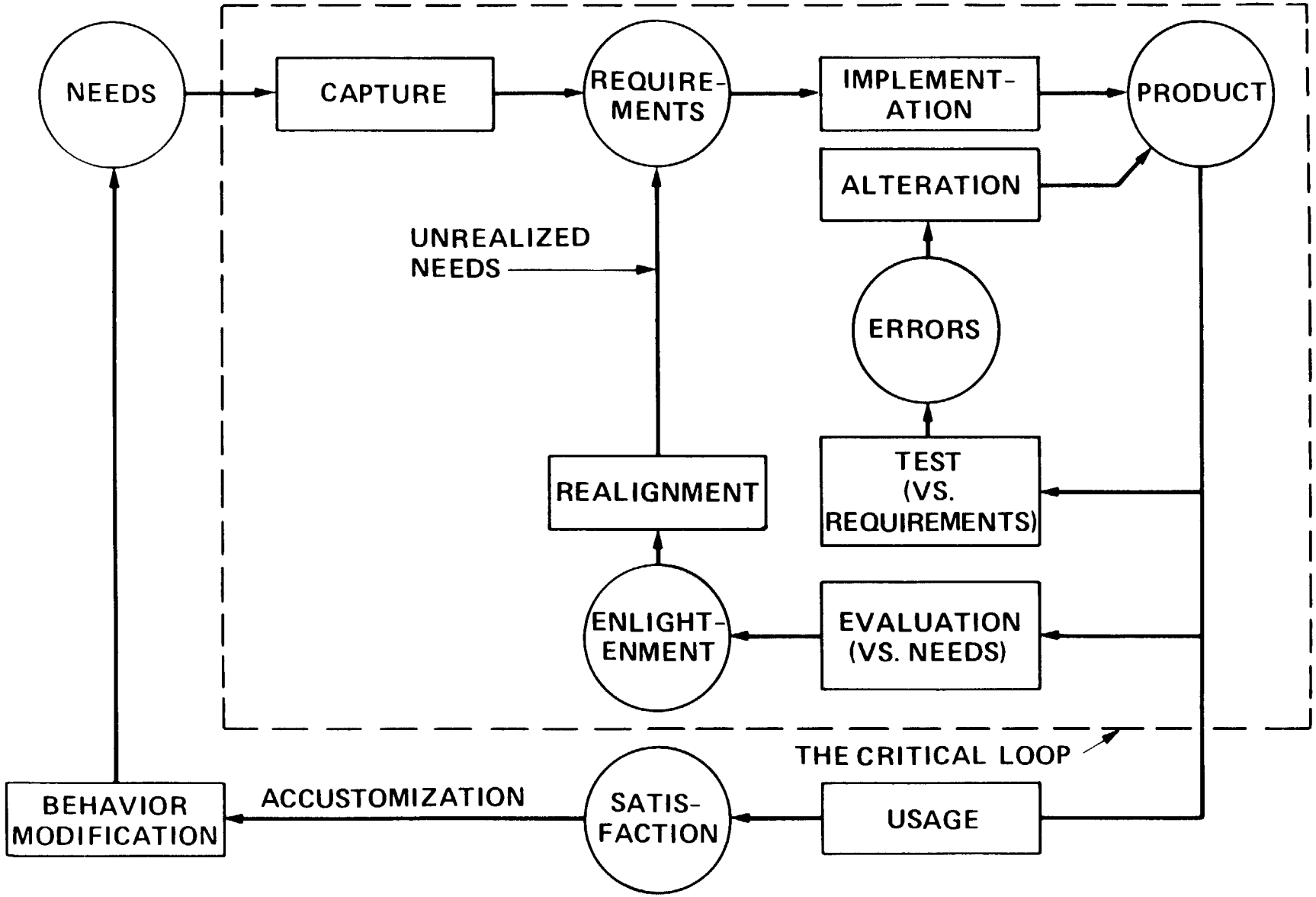
JEFF ULLMAN, CRITICIZING
A RECENTLY PUBLISHED BOOK

JPL COMMUNICATION CHANNEL PROBLEMS

- CHANNEL AND PRODUCT COSTS ARE TOO HIGH
- NEED-TO-SATISFACTION DELAY IS TOO LONG
- NEED-TO-SATISFACTION CORRELATION IS TOO LOW AND DIFFICULT TO COMPUTE
- INPUT AND OUTPUT (I.E., NEEDS AND SATISFACTION) ARE NOT ENTIRELY QUANTIFIABLE, CONSISTENT, NOR TANGIBLE
- COST, DELAY, AND PRODUCT ARE NOT ENTIRELY PREDICTABLE OR CONTROLLABLE
- METRICS FOR MEASURING TRUE INFORMATION CONTENT ARE ONLY APPROXIMATE: LINES OF CODE, FUNCTION POINTS, TOKEN LENGTH, ETC.

A REFINED CHANNEL MODEL

JPL



EACH BOX REPRESENTS A CHANNEL

RCT-5
11-30-88

JPL

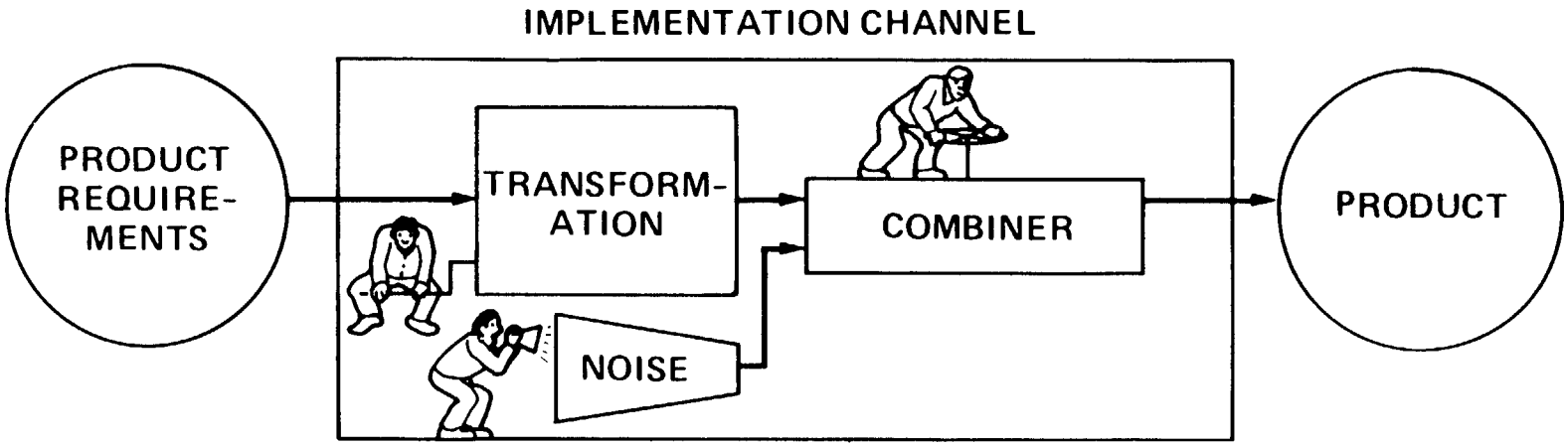
COPING WITH NOISY CHANNELS
(COMMUNICATION THEORY IN A NUTSHELL)

- MEASURE AND CHARACTERIZE THE CHANNEL PARAMETERS
- EXPECT TRANSMISSION TO BE NOISY AND DELAYED
- DESIGN CHANNEL THROUGHPUT TO BE BELOW CHANNEL CAPACITY

$$C_0 = B \log_2 \left(1 + \frac{S}{N}\right) \qquad \text{(SHANNON)}$$

- FORMULATE INFORMATION TO BE RESILIANT TO CHANNEL DISTURBANCES (ENCODING/DECODING)
- TRANSMIT INFORMATION WITH AS GREAT A SIGNAL FORCE AS POSSIBLE
- DO WHATEVER IS POSSIBLE TO REDUCE CHANNEL DISTURBANCES
- USE FEEDBACK TO CORRECT PROPAGATION ERRORS, AS FEASIBLE
- IF YOU CAN'T CONTROL THE CHANNEL, YOU CAN'T INCREASE ITS CAPACITY

FOCUS EXAMPLE

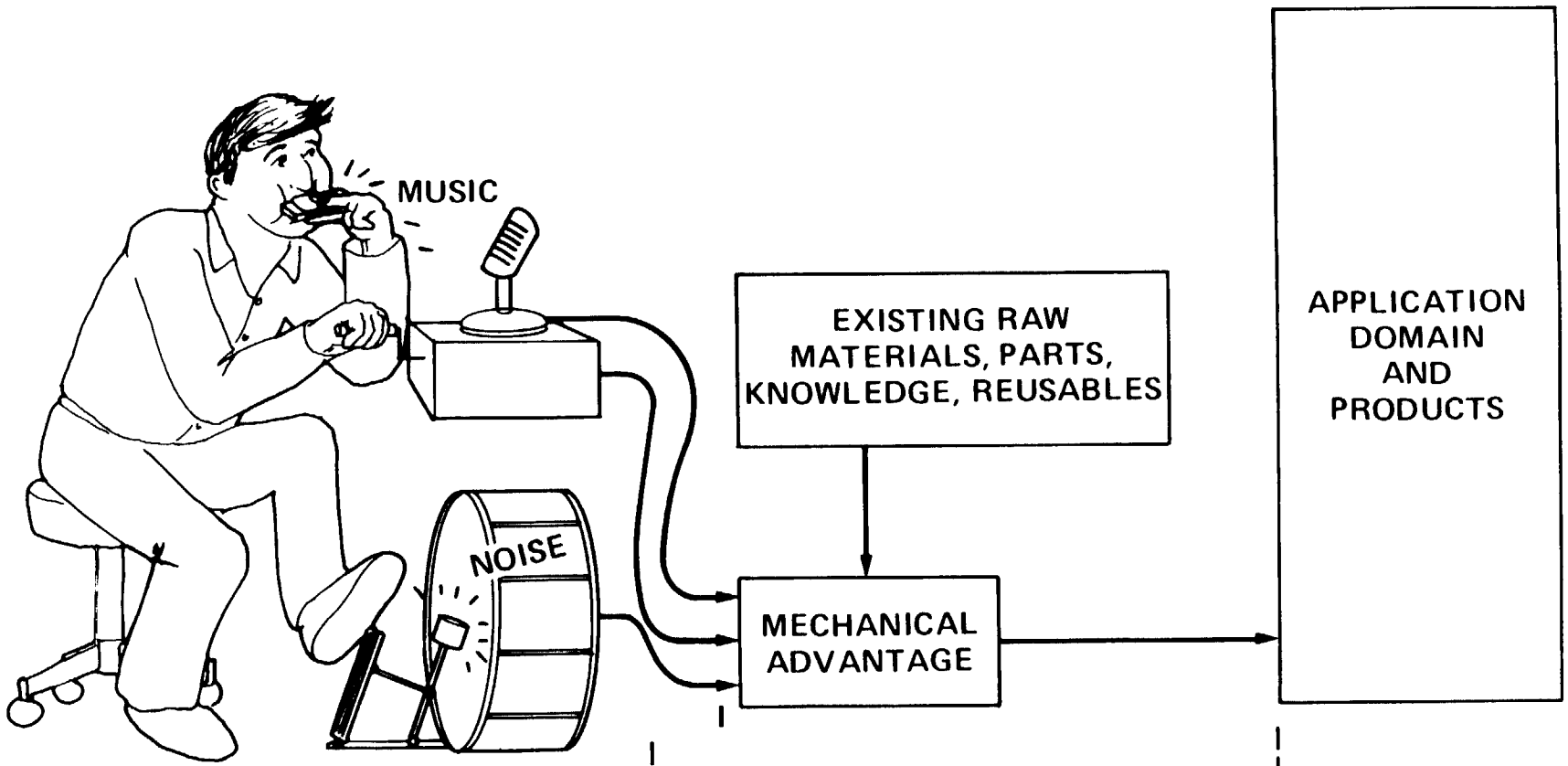


AXIOMS:

- THERE EXISTS A MAPPING OF INPUT REQUIREMENTS TO A PRODUCT SATISFYING THOSE REQUIREMENTS
- INFORMATION CONTENT OF THE REQUIREMENTS IS EQUIVALENT TO THE (CORRECT) PRODUCT SPECIFICATION INFORMATION CONTENT (i.e., THE TRANSFORMATION IS INFORMATION-PRESERVING)
- HUMAN INTELLIGENCE IN THE CHANNEL CONTRIBUTES TO TRANSFORMATION AND NOISE ENERGY, NOT TO PRODUCT REQUIREMENTS INFORMATION
- PRODUCT FUNCTION YIELD Y* RESULTS FROM I* AND I* OF REUSED PARTS
- WHAT WE GET PAID FOR IS THE INFORMATION WE PRODUCE AND THE QUANTITY AND QUALITY OF THE RESULTS IT PRODUCES

JPL

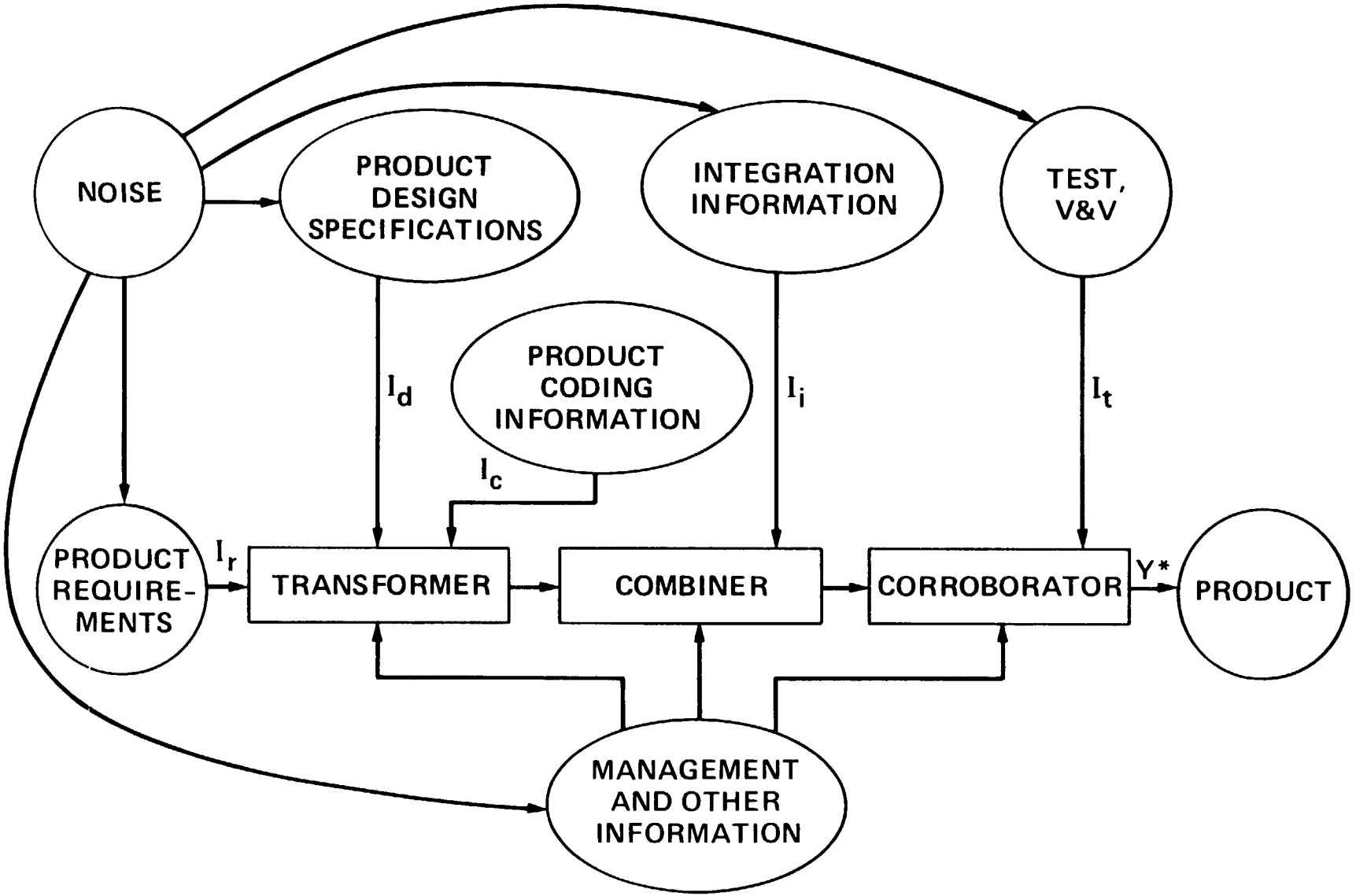
PHYSICAL LIMITS: HUMAN CHANNEL CAPACITY



RCT-8
11-30-88

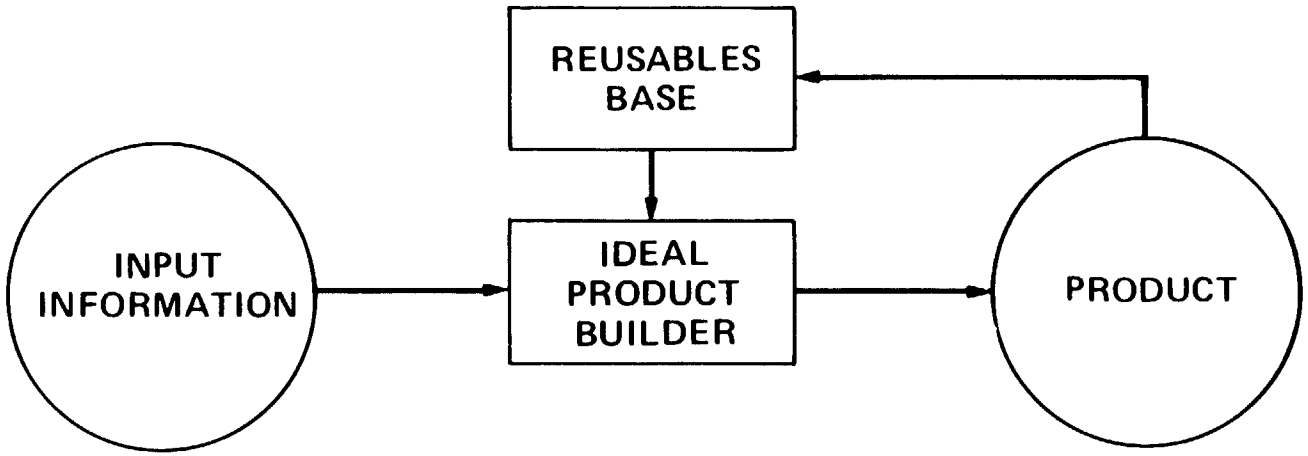
JPL

THE IMPLEMENTATION CHANNEL



RCT-9
11-30-88

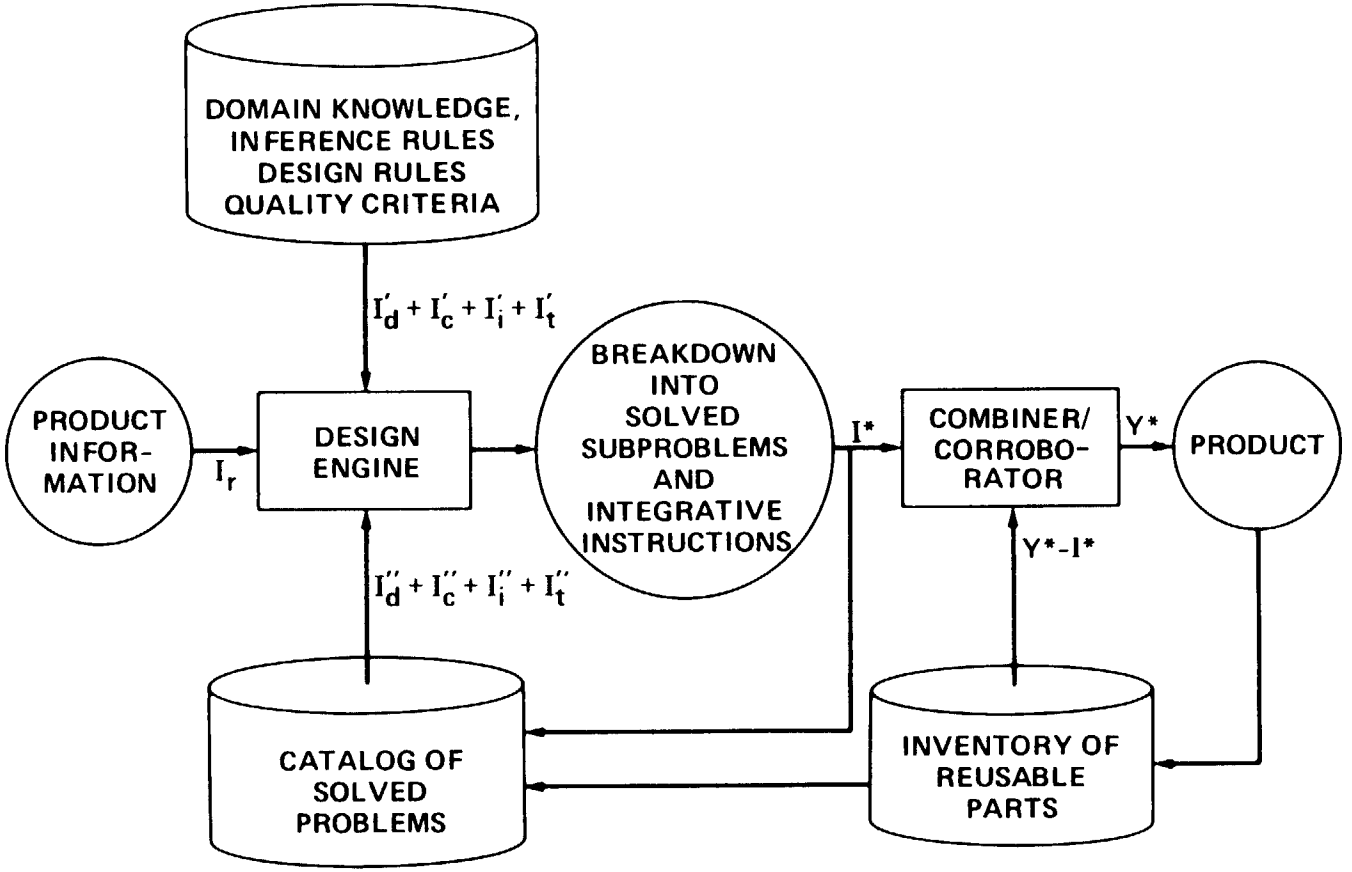
JPL INCREASING THE MECHANICAL ADVANTAGE



- INPUT INFORMATION MUST BE SUFFICIENT TO SPECIFY THE PRODUCT
  - SELECT REUSABLES
  - SPECIFY INPUTS AND OUTPUTS OF EACH
  - INTEGRATE THEM PROPERLY
- THERE THUS EXISTS A MAPPING

  INPUT INFORMATION → PRODUCT → UNIQUE PRODUCT IDENTIFIER
- MAPPING MAY BE MANY-TO-ONE, BUT THE SAME AMOUNT OF INFORMATION BITS GO IN AS COME OUT IN THE PRODUCT (IF BUILDING IS LOSSLESS)
- MECHANICAL ADVANTAGE IS IN PRODUCT YIELD, SO MORE DATA BITS MAY COME OUT THAN GO IN (SUPPLIED BY THE REUSABLES BASE)

THE PRODUCT-BUILDER CHANNEL MODEL



- AMOUNT OF INFORMATION INTO DESIGN ENGINE AND FACTORY ARE THE SAME (ASSUMING NO LOST REQUIREMENTS), BUT IS TRANSFORMED
- KNOWLEDGE BASE AND CATALOG INPUTS REPRESENT TRANSFORMATIONAL INFORMATION USED BY THE DESIGN ENGINE
- IF THESE ARE SUPPLIED BY AUTOMATION, THE PRODUCTIVITY INCREASES

JPL

CHANNEL CAPACITY, PRODUCTIVITY, AND MECHANICAL ADVANTAGE

DEFINITIONS:

W = WORK EFFORT, WORK-MONTHS

I = INFORMATION INPUT, BITS

I* = ESSENTIAL INFORMATION NEEDED TO SPECIFY PRODUCT, BITS

Y = FUNCTION YIELD, BITS

$Y^* = I^* + I^*_{REUSED}$

C = CHANNEL RATE, BITS/WORK-MONTH

P = PRODUCTIVITY, PRODUCT YIELD (BITS)/WORK-MONTH

RELATIONSHIPS:

$$C = \frac{I}{W} \qquad P = \frac{Y^*}{W} = C\frac{Y^*}{I}$$

THE AVENUES TO IMPROVE P ARE THUS IN C, Y*, AND I

JPL

PRODUCTION CAPACITY

- LET W_O BE THAT PART OF W NOT USED TO LOCATE EXISTING INFORMATION, CAPTURE NEW IDEAS, AND PREPARE THESE FOR USE
- LET W_{MIN} BE THE CONCEPTUAL LEAST VALUE OF W_O OVER THE SOFTWARE INDUSTRY

$$P = \frac{Y^*}{W} = \left(\frac{I}{W_{MIN}}\right)\left(\frac{W_{MIN}}{W_O}\right)\left(\frac{I^*}{I_r}\right)\underbrace{\left(\frac{I_r}{I}\right)\left(\frac{W_O}{W}\right)}\left(\frac{Y^*}{I^*}\right)$$

$$= C_O \qquad \mu \qquad \rho \qquad \tau \qquad (1 + \lambda)$$

- NOTE: μ, ρ, τ ARE LESS THAN UNITY BY SHANNON'S LAW. C_O IS FIXED IN THE INDUSTRY, A HUMAN CHANNEL LIMIT
- C_O, μ, ρ, τ, AND λ ARE MEASUREABLE, AT LEAST IN RELATIVE TERMS

JPL

SIGNIFICANCE

- C_O IS FIXED BY HUMAN CHANNEL CAPACITY
- $\mu = (W_{MIN}/W_O)$ IS COMPLEXITY, EXPERIENCE, AND ENVIRONMENT RELATED: IT IS A MENTAL EFFICIENCY
- $\rho = (I^*/I_r)$ IS RELATED TO SUPERFLUITY OF REQUIREMENTS, AND IS LANGUAGE, STYLE, AND AUDIENCE DEPENDENT
- $\tau = \left(\frac{I_r}{I}\right)(W_O/W)$ IS TOOL AND METHODOLOGY RELATED: LABOR-SAVING AND INFORMATION - REDUCTIVE MEASURES
- $\lambda = \frac{Y^* - I^*}{I^*}$ IS THE LANGUAGE ADVANTAGE, MEASURING THE REUSEABILITY REPERTOIRE
- $P < C_O(1 + \lambda)$. THE DEGREE OF REUSE WILL ULTIMATELY DETERMINE THE BOUND ON P

JPL

INCREASING PRODUCTIVITY

- SKILLS (INCREASE μ)
  - EDUCATION AND TRAINING
  - EXPERIENCE
  - STAFF SELECTIVITY
  - SPECIALIZATION
- METHODOLOGY (INCREASE μ, ρ, τ)
  - BETTER FOCUS ON PROBLEM AND ITS SOLUTION
  - MORE EFFICIENT "CHUNKING" OF THE PROBLEM
  - REDUCTION IN LABOR AND INTELLECTUAL STEPS
  - REDUCTION IN ERRORS COMMITTED
- TOOLS AND AIDS (INCREASE μ, τ)
  - REPLACE LABOR AND INTELLECTUAL STEPS
  - REPLACE ERROR-PRONE OPERATIONS
  - RELIEVE LEVEL OF EXPERTISE NEEDED
- SYSTEMS (INCREASE μ, λ, τ)
  - ADD TO KNOWLEDGE BASE
  - ADD TO PRODUCT BASE
  - REUSE COMPONENTS OF KNOWLEDGE BASE AND PRODUCT BASE

JPL

LANGUAGE ADVANTAGE TRENDS

- NOTATION

  R = NUMBER OF UNIQUE OPERATORS OF THE REUSE REPERTORY

  r = NUMBER OF UNIQUE OPERATORS USED

  d = NUMBER OF UNIQUE DATA OPERANDS USED

  N = NUMBER OF ACTUAL OPERATORS AND OPERANDS USED

  $\bar{y}_m$ = AVERAGE YIELD OF OPERATOR MODULES USED

  ζ = (ZiPF ESTIMATE OF N)/N

- THE LANGUAGE ADVANTAGE IS THEN

$$\lambda = \frac{\zeta_r \bar{y}_m}{(r+d)\,[\gamma + \mathrm{LOG}\,(r+d)]\,\mathrm{LOG}_2\,(R+d)}$$

$$< \frac{\zeta \bar{y}_m \,\mathrm{LOG}\,2}{\mathrm{LOG}\,r\,\mathrm{LOG}\,R} < k\,\frac{\bar{y}_m}{(\mathrm{LOG}\,r)^2}$$

- λ CAN CONTINUE TO GROW ONLY IF THE AVERAGE YIELD OF REUSEABLE PARTS CAN BE MADE TO INCREASE FASTER THAN $(\mathrm{LOG}\,r)^2$

JPL

SUMMARY

- THERE IS A PRODUCTION CAPACITY BOUND
- AREAS FOR SOFTWARE PRODUCTIVITY IMPROVEMENTS ARE LIMITED
  - HUMAN/SYSTEM COMMUNICATION CAPACITY
  - MENTAL EFFICIENCY
  - REQUIREMENTS EFFICIENCY
  - TOOL EFFICIENCY
  - LANGUAGE ADVANTAGE
- LANGUAGE ADVANTAGE GROWS IF REUSEABLE MODULES NATURALLY INCREASE IN SIZE FASTER THAN LOG r LOG R (AN OPEN ISSUE)
- PERCEPTIONS ARE THAT CURRENTLY

$$10^{-3} < \mu < 10^{-1}$$

$$10^{-2} < \rho < 10^{-1}$$

$$10^{-2} < \tau < 10^{-1}$$

$$10 < \lambda < 100$$

- THE FACTORS ARE MEASURABLE. INFORMATION THEORY MEASURES NEED TO BE APPLIED TO CALIBRATE OUR INDUSTRIAL BASELINE

KNOWLEDGE-BASED ASSISTANCE IN COSTING THE SPACE STATION DMS

N91-10611

Troy Henson and Kyle Rone

IBM Corporation
3700 Bay Area Blvd
Houston, TX 77058

157

IU196296

ABSTRACT

The Software Cost Engineering (SCE) methodology developed over the last two decades at IBM Systems Integration Division (SID) in Houston is utilized to cost the NASA Space Station Data Management System (DMS). An ongoing project to capture this methodology, which is built on a foundation of experiences and "lessons learned", has resulted in the development of an internal-use-only, PC-based prototype that integrates algorithmic tools with knowledge-based decision support assistants. This prototype SCEAT (Software Cost Engineering Automation Tool) is being employed to assist in the DMS costing exercises. At the same time, DMS costing serves as a forcing function and provides a platform for the continuing, iterative development, calibration, and validation and verification of SCEAT. The data that forms the cost engineering database is derived from more than 15 years of development of NASA Space Shuttle software, ranging from low criticality, low complexity support tools to highly complex and highly critical onboard software.

INTRODUCTION

Software cost engineering (SCE) is the systematic approach to the estimation, measurement, and control of software costs on a project. This discipline provides the vital link between the concepts of economic analysis and the methodology of software engineering. The tasks involved in software cost engineering are complex, and individuals with the knowledge and skill required are scarce (1). The accuracy and consistency of the SCE results are often questionable (2). There is a definite need for tools to enable SCE by managers and planners who are not experts and to improve the results (3).

PROBLEM DESCRIPTION

Software costing is required for the Space Station Data Management System, as in other projects, in many situations. Often the costing is needed within a limited time frame for a proposal, to build a business case, or to evaluate a project that is in trouble or potentially may have a problem meeting cost and schedule constraints if not adjusted. Quantitative es-

timates are required; however, little solid information may be available. A detailed analysis of the software requirements may take weeks if not months. Also, there may be a geniuine concern about how well the software requirements are defined and how stable are those requirements.

To further complicate the situation the estimation process itself carries some inherent risks. Some of the factors that increase risk are software size, complexity and criticality.

Software size, particularly in a system such as the Space Station DMS, is an important factor that can ultimately affect the accuracy of the cost estimate. As the project size increases the interdependency among various elements of the software increases. Problem decomposition, an important step in the costing process, becomes more difficult.

Complexity, i.e., the relative difficulty of the software application, is an important factor affecting development costs. Some types of software are inherently more difficult to develop than others, e.g., development of an operating system compared to the development of utility software. The type of software function, such as real-time, input/output, batch, or computational, and the level of difficulty of the requirements also significantly influence software complexity.

The criticality of the software directly affects the cost of validation and verification as well as indirect costs. Software for certain medical diagnosis or treatment systems, for air traffic control, or for the Space Shuttle Flight Control System must not fail or human lifes will be lost. In contrast, an inventory control system should not fail, but the impact of the failure would not result in the loss of human life.

Viable software costing depends on a quantitative historical database. If no historical data exists, the cost estimation rests on a very shaky foundation. For Space Station DMS, as for other IBM SID Houston projects, the cost engineering database is based on more than 15 years of development of NASA Space Shuttle software, ranging from low criticality, low complexity support tools to highly complex and highly critical onboard software (4), (5).

KNOWLEDGE-BASED SCE AUTOMATION -- SCEAT DEVELOPMENT

Currently at IBM SID in Houston, software cost engineering tasks are performed by a domain expert using his/her experience and data compiled from previous efforts. For a software costing exercise, the domain expert may use stored data and algorithmic/model-based, costing programs; but a significant part of the process is based on non-automated expertise. Software costing expertise is needed in many situations, and the costing is often needed within a limited time frame. Yet, individuals with the knowledge and skill to conduct a software costing exercise are scarce. The knowledge-based decision support assistants in SCEAT identify and preserve the domain experts' knowledge, assist managers and planners who are not costing experts, and improve the accuracy and consistency of the cost estimation results.

As part of the knowledge acquisition process, the first draft of a software cost engineering workbook has been written and utilized as high-level requirements for SCEAT. The overall SCE process was analyzed from a modular/structural/dependencies viewpoint. Included is the relationship of SCE methodologies to other parts of software/systems engineering process control, at one end of the spectrum, and the decomposition of SCE into component tasks and the identification of the SCE foundation or central core, at the other end of the spectrum (See Figures 1 and 2). Then a concise approach to software cost estimation, which covers the total costs -- direct and indirect -- over the complete life cycle, using existing methodologies and tools and quantification of the primary domain expert's knowledge (6), (7) was defined. The experience-based tasks in the SCE process were identified, and the functional design of SCEAT includes expert systems to assist in those tasks. The core development cost estimation methodology was defined in the SCE workbook in more detail and implemented in the initial SCEAT prototype, which includes prototypes of expert systems for assistance in determining software criticality and software complexity.

The SCEAT prototype integrates, under Professional Work Manager (PWM) and EZ-VU on a PC, algorithmic SCE tools with expert systems for decision support assistance. SCEAT integrates the decision support assistant expert systems for software criticality and complexity determination and "stubs" for four additional planned expert systems with nine algorithmic tools including the Matrix Method tool implemented in Lotus 1-2-3. The user interface is via panels offering cook book steps to proceed through the SCE task, selectable information and tools, help screens, and pop-up screens.

COSTING THE SPACE STATION DMS UTILIZING SCEAT

The SCEAT prototype has been utilized to assist in the costing of the Space Station Data Management System (DMS), a complex software system involving a distributed environment with multiple languages and applications (8), (9). The DMS for Space Station is also affected by the requirements for long lifetime, permanent operations, remote integration, and phased technology insertion of productivity tools, applications, expert systems, etc. Major cost drivers include the large size and diversity of the software, complexity, development support environment, off-the-shelf and reusable software, and criticality, which varies from one module to another. An example of the type of results -- at the end of the intermediate step of development cost estimation -- obtained with SCEAT for the DMS costing is included in the presentation.

SUMMARY/CONCLUSIONS

The software cost engineering methodology employed by the domain experts at IBM SID Houston has been captured and integrated into a prototype tool SCEAT (Software Cost Engineering Automation Tool). This PC-based tool integrates algorithmic tools with expert systems which serve as decision support assistants.

SCEAT has been employed to assist in the costing of the Space Station DMS (Data Management System). It is providing a standardized approach for the DMS costing, which involves several individuals. It has made the costing process more efficient and has relieved the demands on the principal domain expert's time, allowing him to move forward into other areas of software/systems engineering process control improvement. The automation and captured methodology domain knowledge has established the foundation and mechanism enabling the continuing calibration and improvement in accuracy and consistency for Space Station DMS costing.

Plans for the future include developing additional knowledge-based decision support assistants and a tutorial to accompany the next version of SCEAT. The approach is also being expanded to other areas of software/systems engineering process control, starting with quality estimation, scheduling and management, and eventually extending to management of performance, product, resources, risk, planning, schedule. (See Figure 1). This is a continuation of the effort to accomplish the long range objective which is to automate, including the development and utilization of knowledge-based systems to serve as decision support assistants, software and systems engineering process control. Results will continue to be applied to assist in the costing and management of the Space Station Data Management System (DMS).

REFERENCES

1. DeMarco, T., Controlling Software Projects, Yourdon, New York, 1982.
2. Kemerer, C. F., "An Empirical Validation of Software Cost Estimation Models," COMMUNICATIONS of the ACM, Vol. 30, No. 5, May 1987, pp 416-429.
3. Boehm, B. W., "Improving Software Productivity," COMPUTER, Vol. 20, No. 9, September 1987, pp 43-57.
4. Madden, W. A. and K. Y. Rone, "Design, Development, Integration: Space Shuttle Primary Flight Software System," COMMUNICATIONS of the ACM, Vol. 27, No. 9, September 1984, pp 914-925.
5. Spector, A. and D. Gifford, "The Space Shuttle Primary Computer System," COMMUNICATIONS of the ACM, Vol. 27, No. 9, pp 874-901.
6. Rone, K. Y., "A Cost Engineering Overview," class notes and handouts, March 29-30, 1988, IBM SID, Houston, Texas.
7. Rone, K. Y., "Software Engineering Process Control," copy of presentation, IBM FSD, Houston, Texas, 59 pages.
8. Chevers, E., "Avionic Systems Test Beds for Space Station," Plenary Presentation at the JAIPCC (Joint Applications in Instrumentation, Process, and Computer Control), sponsored by the IEEE, ISA, and the University of Houston - Clear Lake (UH-CL), March 12, 1987, UH-CL, Houston, Texas.
9. Heer, E. and H. Lum, "Raising the AIQ of the Space Station," Aerospace America, January 1987, pp 16-17.

THE VIEWGRAPH MATERIALS

FOR THE

T. HENSON PRESENTATION FOLLOW

SOFTWARE/SYSTEMS ENGINEERING
PROCESS CONTROL STRUCTURE

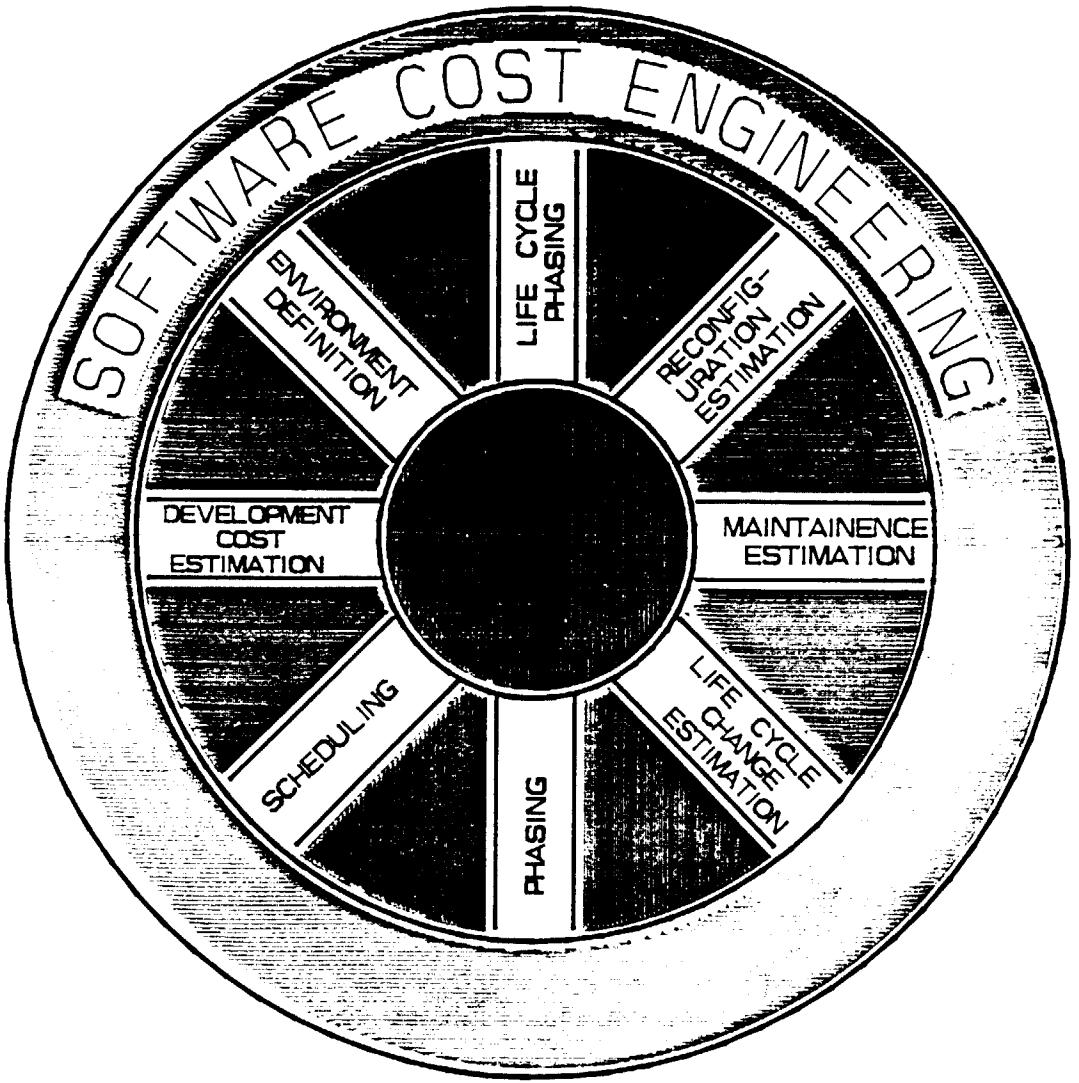
FIGURE 1.

T. Henson
IBM

TFH1ABCD.PIF AT(ORST) 11-28-88

FIGURE 2.

SOFTWARE COST ENGINEERING

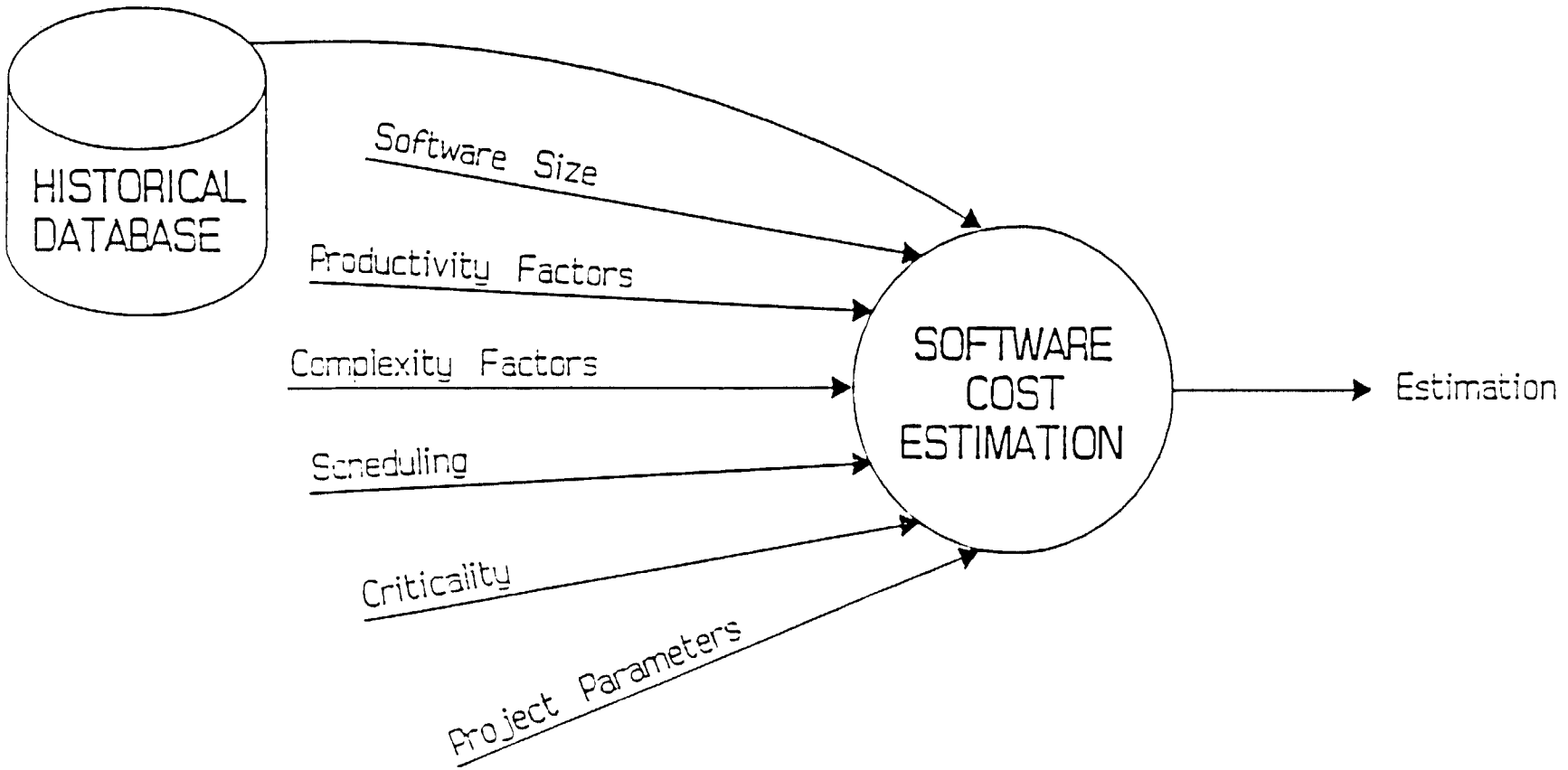


KNOWLEDGE-BASED ASSISTANCE in COSTING the SPACE STATION DATA MANAGEMENT SYSTEM

Troy Henson and Kyle Rone

IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058-1199
PHONE: (713) 282-7476

What Is The Problem ?



KNOWLEDGE-BASED SCE AUTOMATION

- **ACTIVITY DESCRIPTION**
  - Standardize and automate software cost engineering (SCE)
- **OBJECTIVES**
  - Define SCE process
  - Automate SCE
  - Provide SCE courses
- **CURRENT STATUS**
  - Developed SCE courses
  - Developed 9 algorithmic PC tools
  - Developed LOTUS Matrix Method tool
  - Developed criticality and complexity expert systems
  - Developed SCE workbook
  - Integrated tools and expert systems into SCE Automation Tool (SCEAT)
- **FUTURE PLANS**
  - Develop additional decision support assistants for SCEAT
  - Develop a SCEAT tutorial
  - Expand SCEAT into a process control tool

SPACE STATION DMS DRIVERS

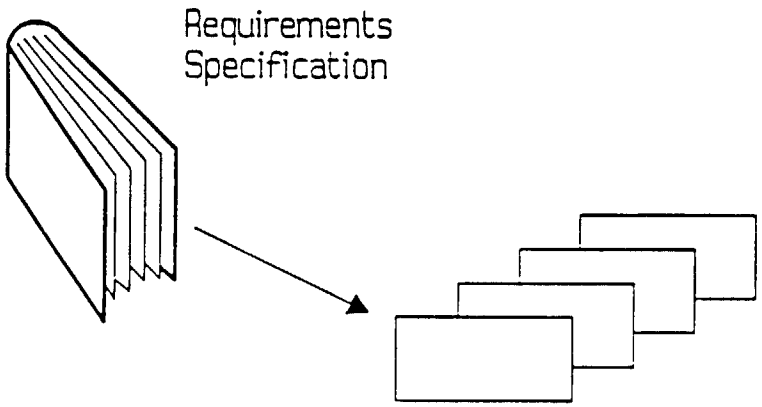
- Long Life
- Permanent Operations
- Remote Integration
- Distributed Environment
- Design-to-Cost System
- Multiple Levels of Criticality
- Phased Technology Insertion
  - Automation (Productivity Tools)
  - Ada
  - COTS
  - Reusability
  - Expert Systems
  - Data Base Technology

DMS COST DRIVERS

- Productivity Levels Driven by
  - Language
  - Complexity
  - Release (Availability of SSE Tools)
  - COTS/Reuse
- Independent Verification Driven by
  - Language
  - Complexity
  - Release (Availability of SSE Tools)
  - COTS/Reuse
  - Criticality
- Indirect Costs Driven by
  - Criticality

COSTING METHODOLOGY

Software Development Cost Estimation



For Each Function

- Estimate Size In SLOCs
- Assign Criticality Level
- Assign Complexity Level
- Assign Functions To Releases
- Select Productivity Factor
- Select Verification Factor
- Select Indirect Factor

→ Calculate Man Months

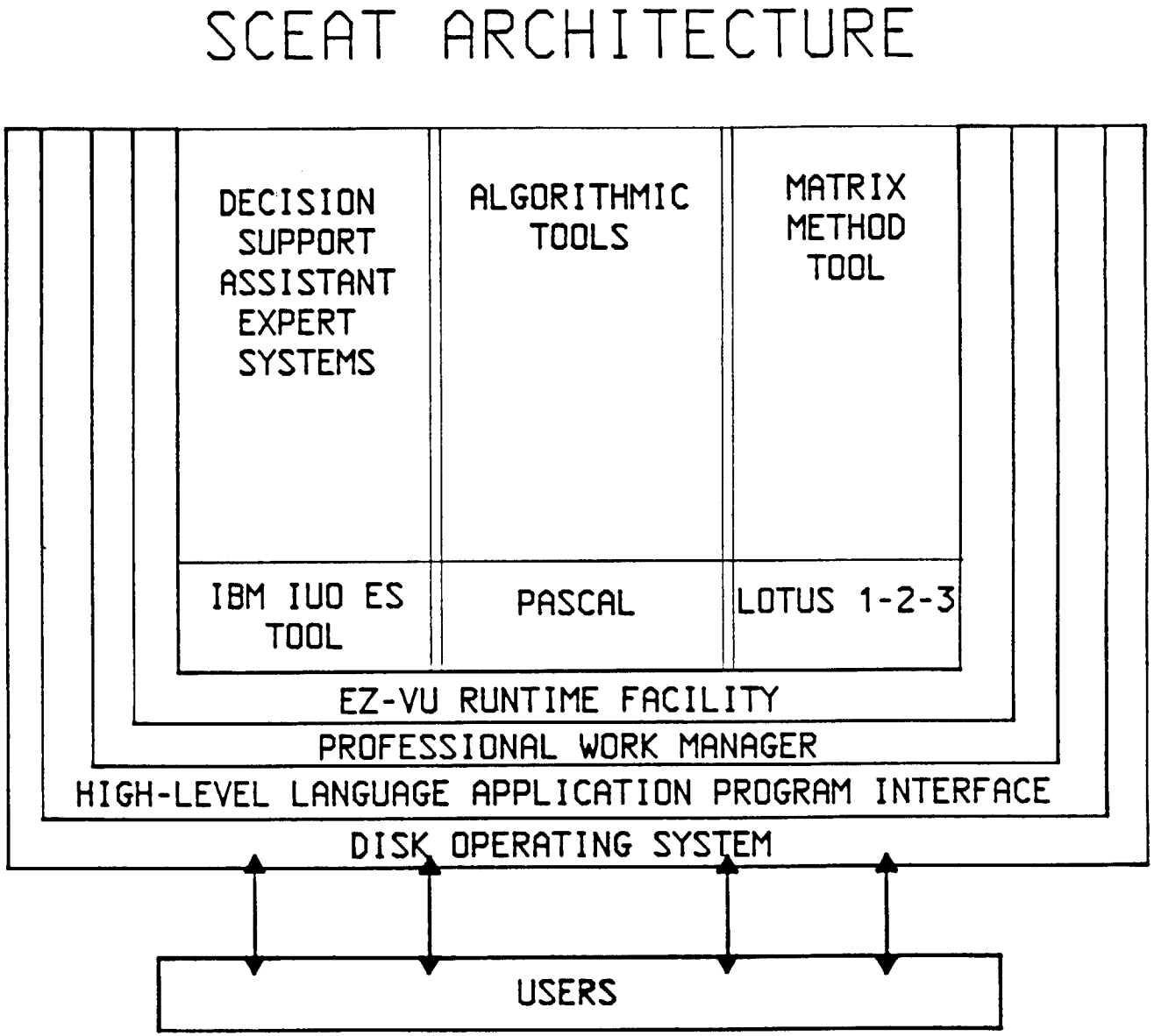
SOFTWARE COMPLEXITY DETERMINATION ASSISTANT ES

- Software Complexity Based on Three Factors
  - Critical constraints
  - Interface requirements
  - Software classification
- Each Factor Assigned a Numerical Weight Range
- Each Factor Considered and Assigned a Numerical Value
- Values are Totaled to Determine the Software Complexity Rating
- Knowledge Captured in the Form of Intelligent Questions

SOFTWARE CRITICALITY DETERMINATION ASSISTANT ES

- Software Criticality Based on Four Factors
  - Human-rated
  - Completion of operational objectives
  - Software distribution requirements
  - Software backup requirements
- Each Factor Assigned a Numerical Weight Range
- Each Factor Considered and Assigned a Numerical Value
- Values are Totaled to Determine the Software Criticality Rating
- Knowledge Captured in the Form of Intelligent Questions

SCEAT ARCHITECTURE



EXAMPLE OF TYPE OF INTERMEDIATE RESULTS

T. Henson
IBM

SUMMARY CHART

AREA	NEW	COTS	LANG.	COMP.	CRIT.	REL.	DEV.	VERIF.	INDIRECT	TOTAL
NOS	59600		ADA	C	SC	1,2,3	202.9	162.2	548.0	913.1
RTOS	7600	40900	ADA	C	SC	1	77.9	62.3	210.6	350.8
Standard Services	4000		ADA	L	M	1,3	10.9	4.4	11.4	26.7
Reconfig. Services	24500		ADA	M	SC,MC,M	1,2,3	78.4	50.6	162.1	291.1
DMS SM	30000		ADA	C,M,L	MC,M,L	1,2,3	115.0	62.8	191.2	369.0
DBMS	22600	50000	ADA	M	M	1,2,3	70.1	28.0	73.6	171.7
OMA	57600		ADA	C,M,L	M	3	101.5	40.6	106.6	248.7
User Services	8000	320000	ADA	M,L	SC,L	1,2,3	121.6	23.5	77.5	222.6
Recon Data Base	100000		4GL	M	L	1	210.5	21.1	69.5	301.1
TOTALS:	313900	410900					988.8	455.5	1450.5	2894.3

SUMMARY

- Software Cost Engineering methodology has been captured and integrated in a prototype tool SCEAT.
  - PC-based
  - Integrates algorithmic tools with expert systems which serve as decision support assistants
- SCEAT has been utilized in costing the Space Station Data Management System.
  - Standardized approach
  - Improved efficiency, accuracy and consistency
  - Captured rationale
  - Foundation enabling continuing calibration and improvement
- Will be expanded into other areas of Software/Systems Engineering Process Control.

30 November 1988

Software Sizing, Cost Estimation and Scheduling

William G. Cheadle
Martin Marietta Astronautics Group
Mail Number L0330
Post Office Box 179
Denver, Colorado 80201

INTRODUCTION

The Technology Implementation and Support Section at Martin Marietta Astronautics Group Denver is tasked with software development analysis, data collection, software productivity improvement and developing and applying various computerized software tools and models. The computerized tools are parametric models that reflect actuals taken from our large data base of completed software development projects. Martin Marietta's data base consists of over 300 completed projects and hundreds of cost estimating relationships (CERs) that are used in sizing, costing, scheduling and productivity improvement equations, studies, models and computerized tools.

BACKGROUND

Martin Marietta resolved in 1975 to establish a study effort to investigate the software development process and the understanding of how to plan, schedule, size, and estimate software. The outcome of this analysis was that management decided to develop a company-peculiar parametric software estimating cost, schedule, and manloading model. This parametric model was generated by using actual software development data collected over a number of years. Cost estimating relationships (CERs) were created, project and mix complexity factors were established, and independent variables were quantified. The result was data base-derived software estimating equations for assembly and high-order language software. These equations and our resulting software parametric models have been validated by comparing project sizing, labor actuals, and schedules with PCEM outputs and documenting the results.

DEVELOPMENT APPROACH

During the early years of our data collection, analysis and model requirements generation activities it was decided that Martin Marietta's software parametric models would include the whole software development life cycle from systems requirements through systems test and provide budget and schedule outputs for the four software development organizations that contribute most to software development. These are:

Systems Engineering,
Software Engineering,
Test Engineering, and
Quality.

Our data base collection approach consists of breaking software actuals out by class, type and language.

Classes of software include:

Manned flight
Unmanned flight
Avionics
Shipboard/Submarine
Ground
Commercial

Types of software are:

Systems Software:	Operating systems and executives.
Support Software:	Simulation, emulation, math models and diagnostic software
Applications Software:	Software that solves the customer's problems.

We collected sizing data by programming language. Our software sizing data base library consists of over 5 million Martin Marietta (Denver) developed source lines of code and over 4 million source lines of code developed by other software development companies and organizations.

At Martin Marietta Denver, we are presently gathering detailed sizing information at the function level to provide additional inputs into our computerized sizing model.

An example of this detailed data is a program of 13,830 SLOC (less comments), of which 9,678 (70%) was programmed in FORTRAN IV and 4,152 SLOC was programmed in assembly language. There were also 1,434 data statements. The sizing summary by computer program component (CPC) consists of the following:

Function Name	Assy	HOL	Total SLOC	Data State-ments
a) Executive/Operating System				
System Control	102	275	377	5
Interrupt Handling	655	64	719	1
Interprocessor communcations	75	139	214	0
Initialization	13	35	48	1
b) Operator Interface				
Menu display and automatic generation	0	1,003	1,003	8
Operator prompting and error checking	0	899	899	4
Tabular displays	0	485	485	51
Graphic displays	0	34	34	0
CRT Formatter	0	22	22	0

c) Data Base Manipulation

Data base generation/regeneration	0	232	323	0
File management	203	94	297	1,116
Data storage and retrieval	0	248	248	9

d) Diagnostics, Fault Determination

Sensor diagnostics	104	3,312	3,416	144
Memory diagnostics	396	1,610	2,006	60
CPU diagnostics	2,510	381	2,891	20

e) Hardware Interface

Peripherals	54	0	54	0
Sensor Device	40	595	635	15
Format manipulation and information conversion	0	159	159	0
	4,152	9,678	13,830	1,434

The "interrupt handling" CPC function level breakout reflected these sizing numbers:

Function Name	Assy	HOL	Total SLOC	Data State-ments
Real time interrupt handler (I)	52		52	
Enable/Disable subroutine	5		5	
Real time interrupt handler (II)	10		10	
Keyboard interrupt handler	53		53	
Keyboard handler subroutine	0	50	50	1
Put character	0	14	14	
Disable interrupts routine	8		8	
Enable interrupts routine	10		10	

MS Interrupt handler	79		79	
MSS Interrupt handler	63		63	
Real time interrupt handler	81		81	
STAR PIP interrupt handler	67		67	
ATOD data ready interrupt handler	51		51	
Deuce/STAR threshold data ready interrupt handler	80		80	
	655	64	719	1

The above detailed sizing data along with the cost and schedule information by project provides the input for our detailed analysis and productivity improvement activities.

PARAMETRIC MODELS

The six models described in this paper are all PC-hosted models and trained users carry disks from job site to job site using available compatible PC computers located at the project facilities. These models provide a management capability that has not been available in the past, and there are no subscription costs or mainframe computer delays using these models.

1) Software Parametric Cost Estimating Model (PCEM)

This model provides a method for estimating the total budget, schedule and manloading for a software development activity. The model addresses all phases of software development from systems requirements through systems test. There are two versions of the PCEM model. Version 3.1 reflects MIL-STD-490/483/1679/1521A development. Version 4.0 reflects DOD-STD-2167 and Ada software development.

Description of the Parametric Model

The data based utilized in the Software Parametric Cost Estimating Model (PCEM) consists of "in-house" and "outside" historical software development actuals collected from over 300 completed software development projects.

The data based software projects were separated by "class" and "type" of software. Each class and type has a different complexity and different cost estimating relationships (CERs).

Class of Software

1) Manned space
2) Unmanned space
3) Avionics
4) Shipboard and submarine
5) Ground
6) Commercial

Type of Software

1) Systems Software
2) Applications Software
3) Support Software

Independent Variables

Several independent variables were investigated and the four which were selected and incorporated into the model are summarized below:

1. Lines of Code - The PCEM accepts either source lines of code or machine instructions (object instructions). The amount of functional decomposition performed prior to arriving at a sizing estimate is very important. A great deal of time and analysis is put into reviewing the decomposition so that a good determination of sizing accuracy can be resolved before we input sizing numbers into the PCEM.

2. Project Complexity - Project complexity consists of 14 factors which reflect how well the customer problem is understood and how prepared the contractor is to respond to solving his problem. The factors are weighted and all 14 must be addressed.

1) Requirements Definition
2) Documentation Requirements
3) Experience of Personnel
4) Experience with Equipment/System
5) Amount of Travel Required
6) Language Complexity
7) Interfaces
8) Man Interaction
9) Development Environment
10) Timing and Criticality
11) New or Existing Software
12) Reliability of Test Hardware
13) Testability of Software
14) Operational Hardware Constraints

3. Mix Complexity - The software mix complexity is applied after software sizing has been accomplished. A hundred percent of the identified software lines of code are distributed across the eight mix elements.

The eight elements of mix complexity describe fractions of the total number of source or object instructions, identified by the software engineer.

1) Mathematics
2) String Manipulation
3) Diagnostics, Support Software
4) Data Storage and Retrieval
5) On-line Communcations
6) Realtime Command and Control
7) Man-machine Interaction
8) Systems software

4. Schedule - PCEM determines the optimum schedule and establishes dates for software milestones. The optimum schedule is defined as that period of time when the software can be developed for the least amount of dollars. Costs will increase if the schedule is accelerated, or if it is stretched out beyond the optimum schedule.

With the four independent variables defined along with class and type information, the PCEM can arrive at a total software cost and schedule estimate.

Organizations Included in the PCEM Output:

The PCEM cost equations provide estimates of budget and schedule for the following three software development organizations:

1) Systems Engineering
2) Software Engineering
3) Software Test Engineering

With the information on source or object lines of code, project complexity, mix complexity and user-supplied schedule, the PCEM computerized model can now arrive at the number of manmonths and the schedule required for each of the three software development organizations.

The equations used in the computerized model are arrived at by a multiple regression methodology assessing and analyzing the collected data base information.

Assembly Language and High Order Language CERs

Development Costs

Equation: $Y = a (x_1^{b_1}) \cdot (x_2^{b_2}) \cdot (x_3^{b_3}) \cdot (x_4^{b_4})$

Where Y = Total Number of Manhours (165 hours = 1 M/M)

x_1 = Estimated Number of Source Lines Code

x_2 = Estimated Project Complexity

x_3 = Estimated Mix Complexity

x_4 = Schedule

a = Constant

b_1, b_2, b_3, b_4 = exponents

Budget and Schedule Information is provided by PCEM for both MIL-STD-490/483/1679/1521A and for DOD-STD-2167 Developments:

Version 3.1 (MIL-STD-490/483/1679/1521A)

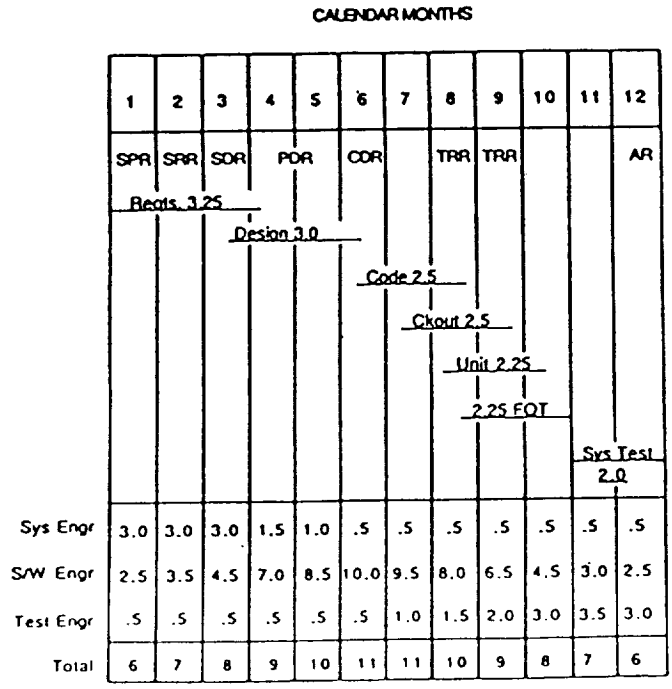
SPR	SRR	SDR	PDR	CDR	TRR	TRR	AR
REQUIREMENTS			DESIGN		CODE		TEST
Systems Reqts	Reqts Alloc	Software Reqts	Prel Design	Detail Design	Code	Checkout	Unit Test	Integration PQT FQT	System Test

Version 4.0 (DOD-STD-2167)

SPR	SRR	SDR	SSR	PDR	CDR	TRR	TRR	FQR
REQUIREMENTS			DESIGN		CODE		TEST		
Systems Concept	Sys S/W Reqts Anal	Software Reqts Anal	Prel Design	Detail Design	Code	Unit Test	CSC Informal Test	CSCI Formal Test	System Integration Test

The computerized PCEM model provides a labor estimate in manmonths, broken out by the phases and subphases of software development. The model identifies an optimum schedule and provides manloading information for each calendar month required for software development. The manmonth estimates are divided between the three organizations that have software development responsibility.

Example Version 3.1:



2. Maintenance Model

The computerized "In Scope" maintenance model was recently validated, and became a Parametric Cost Estimating Model (PCEM) output during the first quarter of 1988. The parametric maintenance model is an historical data based derived tool designed to assist users in estimating the cost of "In Scope" maintenance efforts over a few calendar months or over several years. The software maintenance model output includes those efforts related to maintaining the baseline software configuration through error correction and fine tuning activities.

3. Performance Measurement Model

This state-of-the-art software development performance measurement tool was developed during 1988, and permits independent assessment of on-going software development project performance. The user establishes a performance structure which consists of a list of documentation, design reviews, and milestones that the model is going to use to track software development performance. The model provides a measurement of the performance level based on actuals with respect to budget and schedule and estimates a set of "to complete" budget numbers and calendar months for the identified project. During the course of the development the model identifies where the project is performing at either above or below a 100 percent capability.

4. Sizing Model

The software sizing model is a standalone model which is presently undergoing verification and validation testing, but in the very near future it will become a parametric cost estimating model (PCEM) output. The sizing model provides software development engineers with a new concept computerized functionality software sizing capability. The model gives the user a tool to create software development functional decompositions. Once the decomposition is established, the model helps the user create lower level functional decompositions based on whether the software functional element represents a processing task, an input task, or an output task. Software functionality menus containing generic lists allow the user to indicate functional elements that are components of the software

systems to be developed. As the user identifies software elements, FORTRAN source lines of code estimates are provided by the sizing model. The model also includes an estimating algorithm for data statements sizing.

5. Risk Analysis Simulation Tool (RAST)

RAST is an interactive computer-based application model that provides a technique for performing quantitative software risk assessment. A major feature of the RAST model is the ability to apply statistics to assess cost risk of proposals and on-going projects. The RAST provides the capability to add, subtract, multiply, and divide Monte Carlo derived distributions and constants.

6. Software Architecture Sizing and Estimating Tool (SASET)

This is a new computerized software cost estimating, scheduling and functional sizing model developed for the naval Center for Cost Analysis in Washington, D.C. The SASET model is a forward-chainging rule-based expert system utilizing a hierarchically structured knowledge data base to provide sizing values, optimal development schedules and various associated manloading outputs depending on complexity and other factors. the model is divided in four separate tiers: Tier I, Project Emulation; Tier II, Sizing; Tier III, Complexity; and Tier IV, Maintenance. The model has recently gone through verification and validation testing and the Air Force, along with the Navy, has just recently (September 1988) provided additional dollars to add a calibration enhancement.

ADA

Martin Marietta Denver has been actively involved with the Ada language since its inception. We particpated in the public evaluation of the Red, Blue, Yellow and Green languages before the Green language was selected as Ada in 1979. Over 200 employees have attended our in-house software engineering Ada training course, and over 200,000 SLOC in Ada have been generated by Martin Marietta students and by engineers on projects using the Ada language. In 1981 Martin purchased the NYU Ada/Ed interpreter for the VAX computer and the demand for a higher performance

implementation led to the purchase of a Telesoft/Ada compiler for the VAX/VMS in 1983. Martin Marietta also purchased a validated Rolm Ada Compiler and a Data General Eclipse MV 8000 II computer in 1983. C^3I software developed for a large system started in July 1984 and required rehosting Ada software from the Data General onto a VAX 11/780 computer. During 1987 and 1988 Martin Marietta Denver has won three large command and control projects requiring the use of Ada as the software development language.

CONCLUSIONS

Martin Marietta has one of the largest software development data bases in the country and has been involved in software development data collection, analysis and model building since 1975. Our analysis experts have conducted costing, sizing, scheduling and development management studies on the Ada language for the past several years and have provided new parametric models for Ada management costing and scheduling. Our models and techniques are project tested and geared to providing top management with the tools and resources needed for accurately sizing, costing and scheduling Ada projects and for doing performance measurement on these same projects as they move through the software development process.

C3

THE VIEWGRAPH MATERIALS

FOR THE

W. CHEADLE PRESENTATION FOLLOW

SOFTWARE MANAGEMENT, ESTIMATING, SIZING AND SCHEDULING

PRESENTED BY: W. CHEADLE
MARTIN MARIETTA ASTRONAUTICS GROUP
DENVER DIVISION
MAIL NUMBER L0330
P.O. BOX 179
DENVER, COLORADO 80201

PARAMETRIC MODELS

MARTIN MARIETTA'S DATA BASE DERIVED PARAMETRIC MODELS

- PARAMETRIC COST ESTIMATING MODEL (PCEM) VERSION 3.1
- PARAMETRIC COST ESTIMATING MODEL (PCEM) VERSION 4.1
- MAINTENANCE MODEL
- PERFORMANCE MEASUREMENT MODEL
- SIZING MODEL
- CSCI/CPCI INTEGRATION MODEL
- RISK ANALYSIS SIMULATION TOOL (RAST)
- SOFTWARE ARCHITECTURE SIZING AND ESTIMATING TOOL (SASET)

DATA BASE MARTIN MARIETTA ASTRONAUTICS GROUP

MARTIN MARIETTA DENVER DATA BASE (OVER 300 PROGRAMS)

MARTIN MARIETTA: 29 projects plus 49 other programs
29 projects = 143 programs
192 total

Class of Software: Flight, ground, commercial.

Types of Software: Systems, applications, support.

Languages: HOL (Ada), assembly.

Development Schedule for each Program.

Development Manmonths for each Program.

Organizations Included in Software Development.

Percent of Development Life Cycle.

Source lines of code: 29 projects = 5,026,261 SLOC.

DATA BASE MARTIN MARIETTA ASTRONAUTICS GROUP

OTHER COMPANIES SOFTWARE DEVELOPMENT PROJECTS

Other Companies: 24 projects
24 projects = 110 programs.

Class of Software: Shipboard, ground.

Types of Software: Systems, applications, support.

Languages: HOL (Ada), assembly.

Development Schedule for each Program.

Development Manmonths for each Program.

Organizations Included in Software Development.

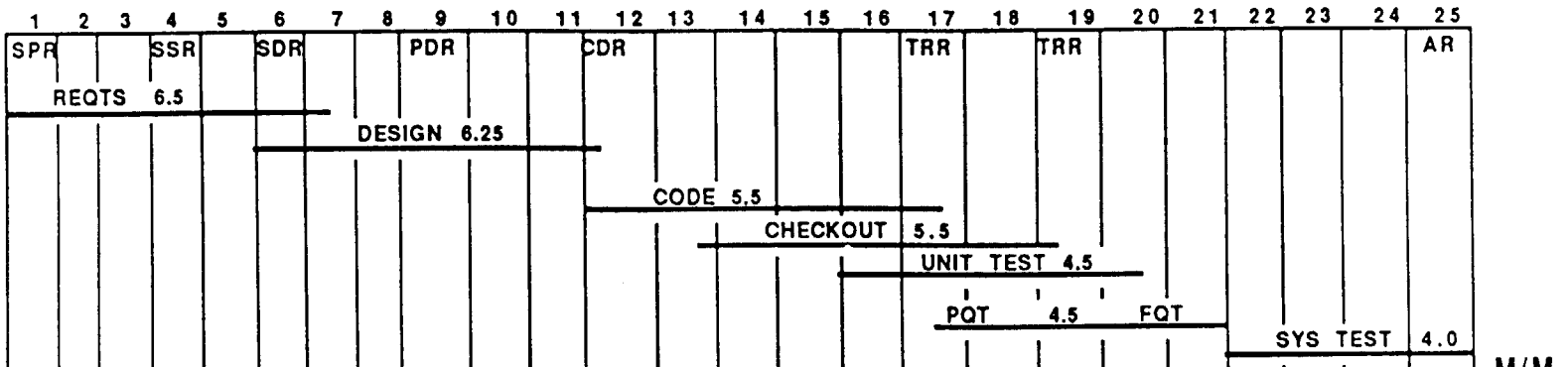
Percent of Development Life Cycle.

Source Lines of Code: 24 projects = 4,282,098 SLOC.

SOFTWARE MANAGEMENT

CLASS OF SOFTWARE : GROUND, NEAR REAL-TIME COMMAND AND CONTROL
CONTRACT TYPE : FPI
PROGRAMMING LANGUAGE : FORTRAN 23,800 NEW SOURCE LINES OF CODE
STANDARD : MIL-STD-1521A
SOFTWARE DEVELOPMENT SCHEDULE : 25 CALENDAR MONTHS
PARAMETRIC COST ESTIMATING MODEL (PCEM) COST AND SCHEDULE ESTIMATE

CALENDAR MONTHS



																										M/M
SYS ENGR	4	4	4	4	4	4	3	3	3	3	2.5	2	1	1	1	1	1	1	.5	.5	.5	.5	.5	.5	.5	50
S/W ENGR	3	4	5	6	7	8	10	11	11.5	11.5	13	14.5	14	13	12	10	10	9.5	9	8	6	5	4	4	4	213
TEST ENGR	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	1	2	2	3	3	3	4	4	5	5	5	5	5	53
QUALITY	.5	.5	.5	.5	.5	.5	.5	.5	1	1	1	1	1	1	1	1	1	.5	.5	.5	.5	.5	.5	.5	.5	17
TOTAL	8	9	10	11	12	13	14	15	16	16	17	18	17	17	16	15	15	14	14	13	12	11	10	10	10	333.0

MARTIN MARIETTA

SOFTWARE DEVELOPMENT

Mil-Stds-490, 483, 1679, 1521A

SPR SRR SDR PDR CDR TRR TRR AR

REQUIREMENTS			DESIGN		CODE		TEST		
System Reqts.	Reqts. Allocation	Software Reqts.	Prelim Design	Detailed Design	Code	Checkout	Unit Test	PQT FQT Integrat.	System Test

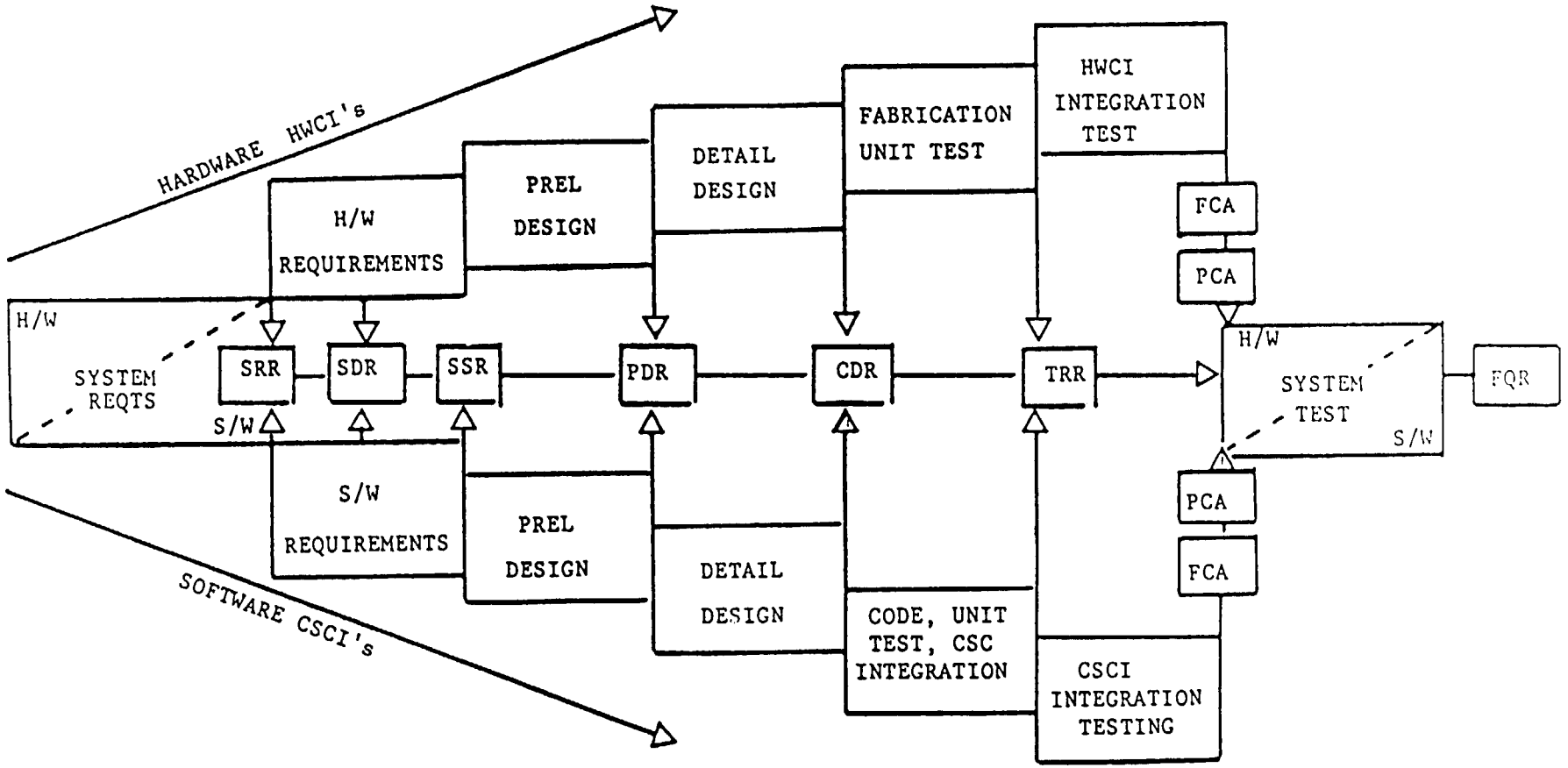
DoD-Std-2167

SPR SRR SDR SSR PDR CDR TRR FCA/PCA FQR

REQUIREMENTS			DESIGN		CODE		TEST		
System Concept	System Software Reqts. Analysis	Software Reqts. Analysis	Prelim Design	Detailed Design	Code	Unit Test	CSC Informal Integrat. Test	CSCI Formal Test	System Integrat. Test

MARTIN MARIETTA

DOD-STD-2167 CONCURRENT HARDWARE AND SOFTWARE DEVELOPMENT



IDEALIZED HARDWARE, SOFTWARE LIFE CYCLE PHASES

SRR	SYSTEM REQUIREMENTS REVIEW	CDR	CRITICAL DESIGN REVIEW	HWCI	HARDWARE CONFIGURATION ITEM
SDR	SYSTEM DESIGN REVIEW	TRR	TEST READINESS REVIEW	CSCI	COMPUTER SOFTWARE CONFIGURATION ITE
SSR	SOFTWARE SPECIFICATION REVIEW	FCA	FUNCTIONAL CONFIGURATION AUDIT	FQR	FORMAL QUALIFICATION REVIEW
PDR	PRELIMINARY DESIGN REVIEW	PCA	PHYSICAL CONFIGURATION AUDIT	CSC	COMPUTER SOFTWARE COMPONENTS

DATA BASE

LANGUAGE RISKS

SYSTEMS SOFTWARE	APPLICATIONS SOFTWARE		SUPPORT SOFTWARE
Machine Instruction	Assembly Language	SPECIAL APPLICATIONS LANGUAGE	Assembly Language
Assembly Language	Jovial	Prolog	Fortran
Pascal	CMS II	LISP	Basic
Fortran	HAL/S		COBOL
Basic	Fortran	5TH GENERATION USER LANGUAGE	Pascal
NEW SYSTEM	COBOL	M204	C Language
ADA	Basic	RAMAS II	NEW SUPPORT
	Pascal	IMS	ADA
	PL/1	Total	TEST SEQUENCE LANGUAGE
	C Language	Ingres	VTL
	NEW APPLICATIONS		STL
	ADA		Comet- H
			GOAL
			HELP
			ATLAS
			SGOS
			CTL

SPAGHETTI CODE DEVELOPMENT

CDR

REQUIREMENTS	DESIGN	CODE	TEST
8%	17%	25%	50%

25% 75%

TOP DOWN STRUCTURED APPROACH (REQTS. DEFINED, DOCUMENTATION EXAMINED AT DESIGN REVIEWS)

SPR SRR SDR PDR CDR TRR PQT FQT AR

REQUIREMENTS	DESIGN	CODE	TEST
23%	22%	20%	35%

45% 55%

USE OF NEW LANGUAGES, TOOLS AND COMPUTERIZED SYSTEMS (ADA, RPS, AI, ...)

CDR

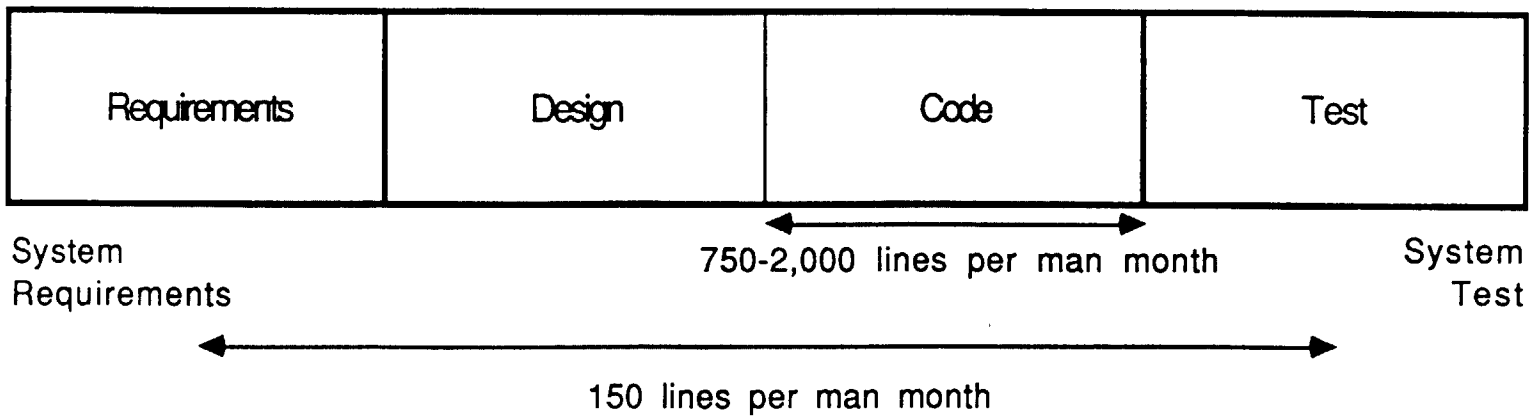
REQUIREMENTS	DESIGN	CODE	TEST
30%	25%	19%	26%

55% 45%

MARTIN MARIETTA

HOURS PER SOURCE LINE OF CODE

- Effort encompasses more than coding.
- Each source line of code represents a unit of effort (work).
- Ground applications software development.



$$\begin{array}{r} .0825 \\ 2{,}000 \,\big/\, 165 \text{ hours} \end{array}$$

$$\begin{array}{r} 1.1 \\ 150 \,\big/\, 165 \text{ hours} \end{array}$$

		Percent
Systems Engr.	.217	14
S/W Engr.	1.1	70
Test Engr.	.25	16
		100%

$$\begin{array}{r} 1.567 \\ .70 \,\big/\, 1.567 \end{array}$$

2.24 hours per SLOC

STUDY APPROACH FOR PREDICTIVE HARDWARE SOFTWARE COST MODELS

- Define Information Collection Requirements and Collect Data.
- Quantitatively and Qualitatively Analyze Data.
- Based on Data Analysis, Develop A Data Base That Will Interface Automatically With A Model.
- Design The Model Using Both Statistical and Quantitative Analysis Techniques.
- Prove The Model By Performance of Validation And Verification Testing.

WHAT CONSTITUTES AN ADA SOURCE LINE OF CODE?

WE CALCULATE ADA SOURCE LINES OF CODE BY COUNTING SEMICOLONS USED AS DELIMITERS, EXCEPT THOSE IN PARENTHESES

NOTE: THIS EXCLUDES SEMICOLONS IN

- COMMENTS
- CHARACTER LITERALS
- STRING LITERALS

ADA SIZING, COSTING, AND SCHEDULING

PER COL. WILLIAM A. WHITAKER:

ADA SOURCE LINES OF CODE ARE CALCULATED BY COUNTING CERTAIN SEMICOLONS.
THERE ARE SEVEN (7) TIMES WHEN SEMICOLONS ARE USED IN ADA.
THREE (3) ARE COUNTED AS SOURCE LINES OF CODE, FOUR (4) ARE NOT.

THE 3 EXAMPLES WHERE SEMICOLONS ARE COUNTED:

1)	SEMICOLONS THAT TERMINATE CLAUSES	WITH TEXT_IO;
2)	SEMICOLONS THAT TERMINATE DECLARATIONS	A : INTEGER;
3)	SEMICOLONS THAT TERMINATE STATEMENTS	C := A + B;

THE 4 EXAMPLES WHERE SEMICOLONS ARE NOT COUNTED:

4) SEMICOLONS THAT TERMINATE PARAMETERS IN A LIST ENCLOSED BY PARENTHESES. (A : INTEGER ; B : FLOAT)
5) SEMICOLONS IN COMMENTS -- TEXT;
6) SEMICOLONS USED IN SINGLE QUOTATION MARKS (CHARACTER LITERALS) ';'
7) SEMICOLONS USED IN DOUBLE QUOTATION MARKS (STRING LITERALS) " A ; B "

MARTIN MARIETTA

ADA SIZING, COSTING, AND SCHEDULING

EXAMPLE ADA PROGRAM	LEGEND
WITH TEXT_10;	E
PROCEDURE EXAMPLE IS	A
	B
--THIS IS A COMMENT; NOT A LINE OF CODE	C
TYPE Z IS RANGE 4 .. 44;	D
CHARACTER_LITERAL: CHARACTER := ';' ;	D
STRING_LITERAL: STRING := " x ; y ";	D
PROCEDURE FIRST IS (R: IN Z; S: OUT Z) IS SEPARATE;	D
BEGIN	A
IF (A = 22) THEN	A
B := 4;	S
END IF;	S
	B
END EXAMPLE ;	D

THIS ADA EXAMPLE PROGRAM CONTAINS 14 CARRIAGE RETURNS

THERE IS	1	COMMENT STATEMENT	C
THERE ARE	3	TEXT LINES	A
THERE ARE	2	BLANK LINES	B
THERE ARE	8	ADA SOURCE LINES OF CODE 5 DECLARATIONS	D
	14	2 STATEMENTS	S
		1 CLAUSE	E

MARTIN MARIETTA

ADA SIZING, COSTING, AND SCHEDULING

ADA LANGUAGE ATTRIBUTES

- STRONG DATA LINKAGE BETWEEN PARENT MODULE & SUBORDINATE MODULES.
- EXCEPTION HANDLING... IN THE EVENT OF AN ERRONEOUS CONDITION, ERRORS WILL BE IDENTIFIED.
- PACKAGES: USED TO GROUP RELATED ENTITIES THAT CAN BE CALLED FROM OUTSIDE THE PACKAGE.
- STRONG TYPING: ENSURES THAT ERRORS ARE DETECTED AT COMPILATION TIME.
- GENERICS: ENCOURAGES RE-USEABILITY, ALLOWS SOME LOGIC STRUCTURE TO BE USED OVER AND OVER.
- TASKING: ALLOWS EVENTS TO BE RUN IN PARALLEL.
- FAULT TOLERANCE: ABILITY OF EITHER H/W OR S/W TO DETECT AN ERROR AND TO RESPOND.

PANEL #3

STUDY OF SOFTWARE PRODUCTS

H. Sayani, Advanced System Technology Corporation
J. Hihn, Jet Propulsion Laboratory
R. LaBaugh, Martin Marietta

REVERSE ENGINEERING

AN AID TO UNDERSTANDING SYSTEMS

Presented

At

The Thirteenth Annual Software Engineering Workshop

N A S A

Greenbelt, MD

November 30, 1988

By:

Hasan H. Sayani, Ph.D.

Advanced Systems Technology Corporation (ASTEC)

9111 Edmonston Road - Suite 404

Greenbelt, Maryland 20770

(301) 441-9036

Reverse Engineering
An Aid in Understanding Systems

by

Hasan H. Sayani, Ph.D.

Advanced Systems Technology Corp.
9111 Edmonston Road, Suite 404
Greenbelt, MD. 20770

1.0 THE NEED FOR REVERSE ENGINEERING

Several reasons may bring an organization to consider reverse engineering. It is possible that an organization's software (code) has not been adequately documented, either from its inception or after multiple rushed changes. To understand the system behavior, or to maintain the system, the organization would need a more global view than that provided by a program. On the other hand, an organization might find that it would like to consider, before actual redesign, the impact of proposed changes to an existing system. Or, an organization might need to grasp how two or more existing systems could be integrated. One other reason might be to update the underlying technology of hardware, operating system or system software (such as change from a file management system to a database management system).

2.0 WHAT IS REVERSE ENGINEERING AND HOW IS IT APPLIED?

The process of Reverse Engineering entails translating existing code into some "higher" form. Reverse engineering can be applied for one of several applications.

2.1 Making Code Easier To Read

When programs have evolved over time, and written by various individuals with differing degrees of sophistication, the resultant program code becomes difficult to read. In such cases, Reverse Engineering may help in re-structuring the code (often referred to as "re-engineering") to make it easier to comprehend.

2.2 Synthesizing Diverse Existing Systems

Previously stand-alone systems may need to be synthesized into a coherent single system. In such cases, the individual systems may have been written in different programming languages, or use different technology to manage data. Reverse Engineering would help in producing a synthesized abstraction which could be properly evaluated for procedural, control and data structure consistency and a new system re-generated from such abstractions.

2.3 Maintaining An Existing System

Making changes to an existing system requires that the maintainer understand the effect of making the changes. In particular, it is important to recognize not only the first order effects but also the ripple effects. The Reverse Engineering mechanism can be used both to estimate the impact of the change and to ensure that the change is made correctly.

2.4 Redesign Of An Existing System

The development of a new system requires that it retain all the desired features of the current system and incorporate the new features. Further, the deployers need to be able to show the relationship of the new system to the existing (old) system. This task is made much easier if the basis for the new system is an abstraction of the current system.

3.0 COMPONENTS OF A REVERSE ENGINEERING TOOL SET

A Reverse Engineering System is made up of several components as shown in the accompanying figure.

3.1 Generalizable Translator

The main component of the Reverse Engineering System is a generalizable translator which has two main parts: one that recognizes known constructs of the language, and another that can perform the appropriate actions desired when a construct is recognized.

3.2 Abstraction Repository

The major action that the generalizable translator performs is the production of abstractions suitable for storage and retrieval. Hence, a required component of a Reverse Engineering System is an interface to an appropriate repository. An example of such a repository is the PSL/PSA system. A key characteristic of such a repository is the availability of a formal underlying conceptual model that is not tied to a specific programming language, and one that permits controlled synthesis of abstractions.

3.2.1 Browsing Capability

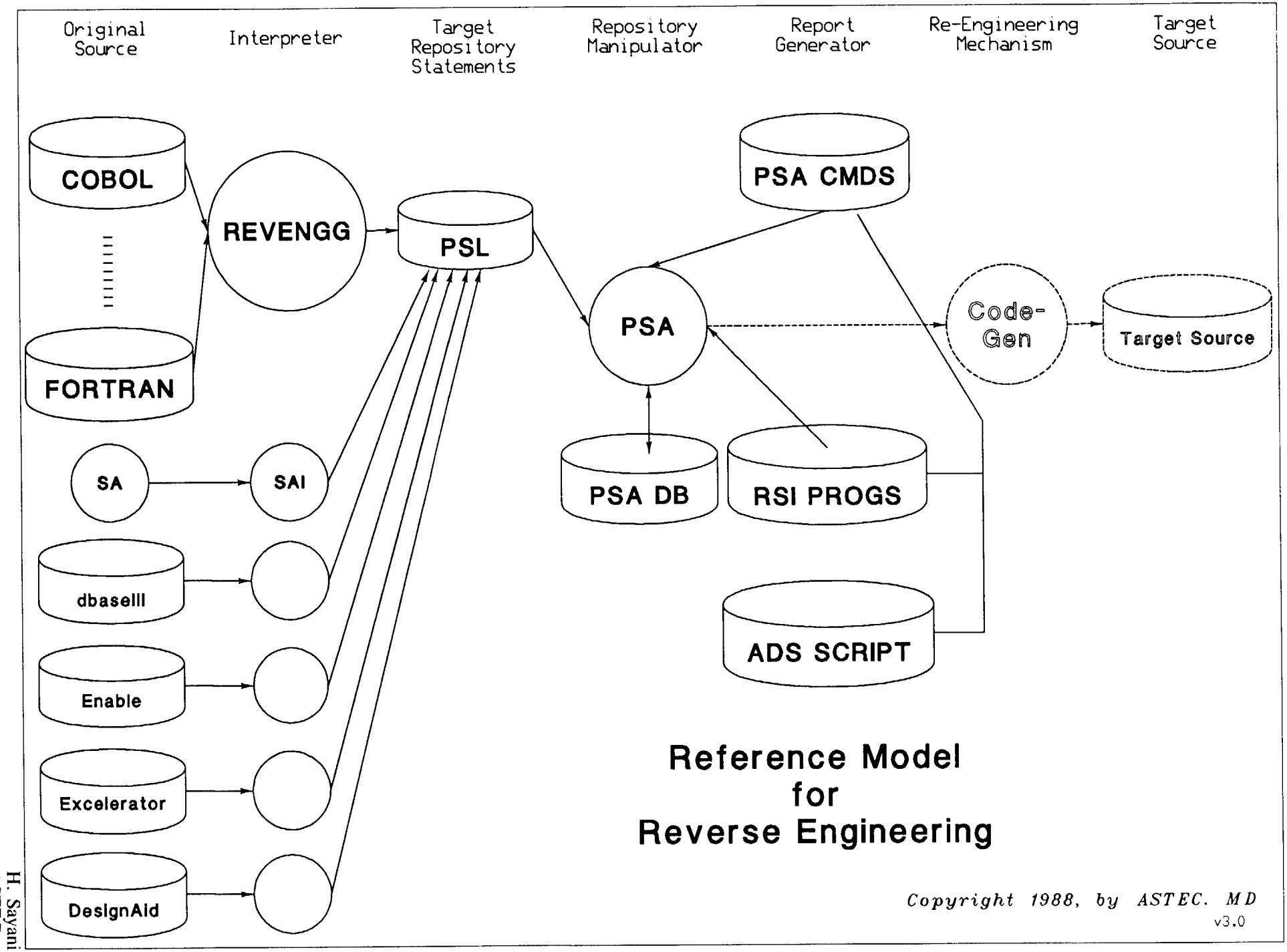
The repository must have capabilities which allow the users to browse/query the repository in a completely flexible fashion.

3.2.2 Reporting Capability

The repository system must have a reporting mechanism that permits the production of reports per specified format, or "download" information that can be input to other tools such as CASE tools.

3.3 Code Re-generation

Some applications may require that the code abstracted be re-generated (if only minor changes have been made). A complete re-development of a system from a higher level abstraction would fall into the category of automated system development and is beyond the scope of the discussion in this paper.


Original Source
Interpreter
Target Repository Statements
Repository Manipulator
Report Generator
Re-Engineering Mechanism
Target Source
COBOL
FORTRAN
REVENGG
PSL
PSA CMDS
PSA
Code-Gen
Target Source
PSA DB
RSI PROGS
ADS SCRIPT
SA
SAI
dbaseIII
Enable
Excelerator
DesignAid
Reference Model for Reverse Engineering
Copyright 1988, by ASTEC. MD
v3.0

4.0 A REVERSE ENGINEERING METHODOLOGY

There are several steps involved in applying the Reverse Engineering process.

4.1 Recognizing The Programming Language Dialect

Since no programming language conforms perfectly to a standard, reverse engineering requires the practitioner to examine the code and identify special coding constructs that deviate from the norm. This implies access to a representative sample and a "pilot" application of the process.

4.2 Accomodating The Identified Programming Language Dialect

The generalizable translator may have to be given additional rules for handing both normal coding constructs and those that are special to this code.

4.3 Translating The Code

The code is then passed through the generalized translator to produce the abstraction that can be entered into the repository.

4.4 Examining the Abstraction

Reports are derived from this mechanism for examination and evaluation. Formal documentation can be produced incorporating this information.

4.5 Using Ancilliary Tools

The information can be passed to another tool (e.g., a CASE graphics package) for viewing the structure and function of the code in pictures.

4.6 Integrating Systems

Information about (an)other system(s) can be merged with the information about the reversed engineered system to determine impact of integration.

4.7 Code Re-generation

Information about the target system could be handed off to a translator for reinterpretation in the form of a programming language.

5.0 SCENARIO OF USAGE

There are several strategies for using the Reverse Engineering Mechanism. The one described below has evolved over several time and takes into account the need to manage large amounts of information and to evaluate the target system in detail as well as in its full scope. The whole process also tends to be iterative.

5.1 Micro Examination

First, individual units of code (e.g. Programs, Copylibs) are translated. Each of these translations are stored as an isolated database in the abstraction mechanism. This permits the examination of local structures: procedural as well as data structures. It also affords an opportunity to examine the algorithm used at a "micro" level.

5.2 Macro Examination by Features

After all the individual units of code have been examined, all those units that comprise a system need to be synthesized. One obvious approach is to take all the individual abstractions (individual databases) and "merge" them together. Experience has shown that such a database becomes far too large and unwieldy, both from the performance standpoint and the human factors. An alternative strategy is to synthesize subsets of individual databases. An example would be to extract all procedural interactions between code units and populate a "procedural structure" database. Another such synthesis would pull out the data structures, and still another might make a detailed "data element dictionary" database. Each of these could be examined and annotated as necessary. This strategy does not preclude eventual merging of these databases into a composite database.

5.3 Evaluating The Abstractions

Both the individual databases and specially synthesized databases can be used in "browse" or "query" mode to pinpoint answers to questions that precipitated the Reverse Engineering process. Answers may be sought for questions about the boundaries of the system, the degree of coupling, the implications of changing data structures, etc.

5.4 Applying The Results

The answers obtained above would make it feasible to take the necessary actions to solve the problem. These actions could result in a strategy of performing certain tasks such as determining the scope of the ripples likely to occur during a particular maintenance task, or a strategy for the addition of other design components using CASE tools and requiring a re-design of the new system.

6.0 USAGE OF REVERSE ENGINEERING ON ACTUAL PROJECTS

Reverse Engineering has been applied to various systems with differing objectives.

6.1 Maintenance Application

A particular application, the maintenance of a complicated information system, will be used as an illustration of the potential payoff for the application of Reverse Engineering.

This system was made up of several subsystems each with many major functions and sub-functions. These sub-functions eventually were broken down into primitive processes (as in Structured Analysis). To understand the magnitude of the problem, one of the subsystems was made up of 45 major functions which broke down into 329 sub-functions which in turn resulted in 2,711 primitive processes. Similarly, one of the components of the system had 36 data stores with 4,498 record types. Several of these record types had over 180 data elements. Finally, to illustrate the maintainers' nightmare, one of the Processes used 64 data elements and changed 61 of them. Similarly, one of the data elements was used by 422 Processes and changed by 455 Processes!

This system was Reverse Engineered for the purpose of maintaining it. Management kept statistics and a semi-controlled parallel group that performed the maintenance task without the aid of Reverse Engineering tools. Maintainers with the Reverse Engineering System reported an 8 to 1 improvement in productivity while noting that certain types of maintenance assignments would not even have been attempted by them had they not had access to the Reverse Engineering Mechanism. The casual statistics from the control group (without the tools) showed that they were still working on the problem four days after being assigned it while the group with tools had fixed it in two hours. Further, the group with the tools had far more confidence in the "fixes" made than the group without the tools. Lastly, the group with the tools was able to estimate the time needed to perform the fix with some degree of confidence after studying the problem whereas the other groups guestimates were off the mark, often by an order of magnitude.

6.2 Re-design Of Existing System

In another application, system developers were able to use the Reverse Engineering Mechanism to quickly understand the "current physical" system. They annotated portions of it with the help of current users, and were able to move on rapidly to add new features desired. This was done with the confidence that they had not left out any of the desired features of the current system.

Based on these and similar projects, we conclude that reverse engineering is feasible and can be invaluable to organizations that:

- have to maintain poorly documented code
- want to redesign a system poorly understood system
- need to project the impact of desired changes to a system
- require the integration of multiple systems.

7.0 POTENTIAL PITFALLS

We would not like to leave the impression that Reverse Engineering is a simple, trivial solution to all problems of managing code. Properly managed and with realistic expectations it can be a most useful approach. However, there are several potential pitfalls that an organization may encounter. They range from very mundane problems of low level technology to subtle issues of organizational politics. We touch upon a few of these below.

7.1 The Mechanics

Low level technology problems are of the type which make it difficult to transfer data (e.g., source code) from the operational system to the Software Engineering Environment in which the Reverse Engineering Mechanism is housed. These range from mismatches in tape formats available and readable, to the introduction of spurious information (or the removal of useful information) in a transfer across a Local Area Network. In two of the projects we were overseeing, this process caused a delay ranging from one to six weeks.

7.2 Local Variations in Programming Languages

Supposedly standard programming languages may have local variations taken care of by local pre-processors. An example of this was a system where we found (by browsing through the repository of abstractions) that several paragraphs in a COBOL program were referred to but were not found in the code translated. We were informed that those were taken care of at "pre-compile" time!

7.3 Stylistic Variations

A system which has evolved over time usually has been worked upon by several programmers. Each of these may have learned particular styles of programming. Further, these styles also evolve over time. However, there never is time to bring previous programs upto date to conform to current styles. Hence, it becomes difficult to comprehend why one program grouped a certain set of operations differently from another program in the same system.

7.4 Lack Of Standards And Conventions

Even today very few development shops have comprehensive standards and conventions for programming. A classic example is the naming of data and procedures. Both the style of naming and the scope of this naming can cause a significant amount of problems when they are being studied as abstractions. For instance a name may be made up of components which may be abbreviated inconsistently. Or, a data name may be qualified by the program it appears in (making it de facto local data) even though it is shared globally, thus making it difficult to synthesize a system-wide view of data. One extreme case of this type was where a database designer had used a distinct name for each data element in every view rendering the database design useless.

7.5 Technology Transfer

Technologists often do not realize the importance of recognizing the effect of commerce on their products. To illustrate, while a technologist would be self-congratulatary about the eight to one savings of costs, a contractor would be concerned about the "cost plus" implications of such a technology! Finally, individuals who have learned to perform tasks such as program maintenance without the use of tools may often feel threatened that much of their expertise would be rendered superflous with these tools. They would be quick to point out the flaws of these tools - after all they were not invented here!

To summarize, the Reverse Engineering Mechanisms we discuss here are not simple, pre-packaged solutions that can be brought in to an organization and by their mere installation provide all the potential benefits. We feel that these tools are better compared to the concept of the big-8 "practice"; i.e., they need to be adapted to the local situation, helped along with consultation and the evolved tool then left behind for use by the organization, if desired.

8.0 FUTURE DIRECTIONS

We feel that the Reverse Engineering concept has barely touched the tip of the development iceberg. We ourselves are interested in several aspects of the process and will highlight some of these below.

8.1 Improved Interpretation Of Source Code

Current approaches use the "compiler-compiler" approach for the interpretation of the code and the performance of actions to be taken when known constructs are recognized. This approach requires "re-binding" of mechanisms every time the simplest of variations has to be made. By its very nature, it requires the tool developer to perform this task. We see this process being replaced by more sophisticated mechanisms which would not only make the task easier, but also allow the end-user to make the selection of actions to be taken. We feel confident that this can be achieved because we have developed this technology and is in use in our bridges to CASE tools.

8.2 Better Repository Interfaces and Abstractions

There are several approaches to translating code to some other form. One is a simple one pass approach which interprets the code; another is a multi-pass translation with internal "symbol table" development; the last pass translates the contents of the symbol table to the desired abstraction. The most desired approach would provide an active interface to a dictionary system which would allow the enriching of a dictionary database as more information became available about an item from the source code. This and the need to regenerate codes in different languages would require the development of a more sophisticated abstract model of programming.

8.3 Better Interfaces To Other Tools

Since the interpretation of code as abstractions results in a complicated information system, it is natural to provide some computer-aided support for browsing through these abstractions. Good repository systems such as PSL/PSA provide this capability. Another natural medium would be CASE tools. Hence, it would be appropriate to perform a translation of code into, say, Structure Charts or Data Flow Diagrams which could not only be examined by CASE tools, but also modified using the CASE tools. This process is becoming feasible now. We feel that such needs will also lead to improvement in curent methodologies for analysis and design and an improvement in the "forward" process of systems development by requiring more precise traceability and standards.

8.4 Re-generation Of Code

While "re-engineering" mechanisms can perform this task today, it is performed in a rather restrictive sense. Usually, the regenerated code is in the same language as the original code, or the translation is an incomprehensible line by line encoding of code from the source language to a target language. The latter approach often results in a "step child" syndrome. The newly generated code is neither understood by the source code specialists nor the target code specialists. We feel that the abstraction to a higher level view and the re-construction to a view specific for a desired target language would be more appropriate. This requires a better abstraction model (as discussed above) that models both the source and target languages and programming in general.

8.5 Technology Transfer

Finally, we have been sobered enough by practical experience of transfering technology to using organizations to realize that the best of technology will only perform to its full potential only if properly introduced. This requires careful handling of issues ranging from politics, human factors, finance and hidden agendas. Some of us feel that the consideration that needs to be given to these factors often outweighs the technology by as much as four to one!

THE VIEWGRAPH MATERIALS

FOR THE

H. SAYANI PRESENTATION FOLLOW

REVERSE ENGINEERING

WHAT IS REVERSE ENGINEERING?

Working back from a phase in the development Life Cycle

- from program code
- to a possible design which the code implements
  - or, to the requirements which the design addresses
  - or, ...

In the absence of supporting documentation, it is akin to

- An Archaeological process
  - " ...we see these hieroglyphics, therefore, ..."

Success depends on:

- recognition of possible:
  - Loss of information
  - Ambiguities
- willingness to:
  - Supplement the information
  - Capture it formally

H. Sayani
ASTEC

REVERSE ENGINEERING

WHY PERFORM REVERSE ENGINEERING?

To understand the current system

- "...Why does the system behave like this when we..."

To be able to make changes to the current system (maintenance)

- "...If we were to change this, what would its impact be?..."

To be able to modify the system (enhancements)

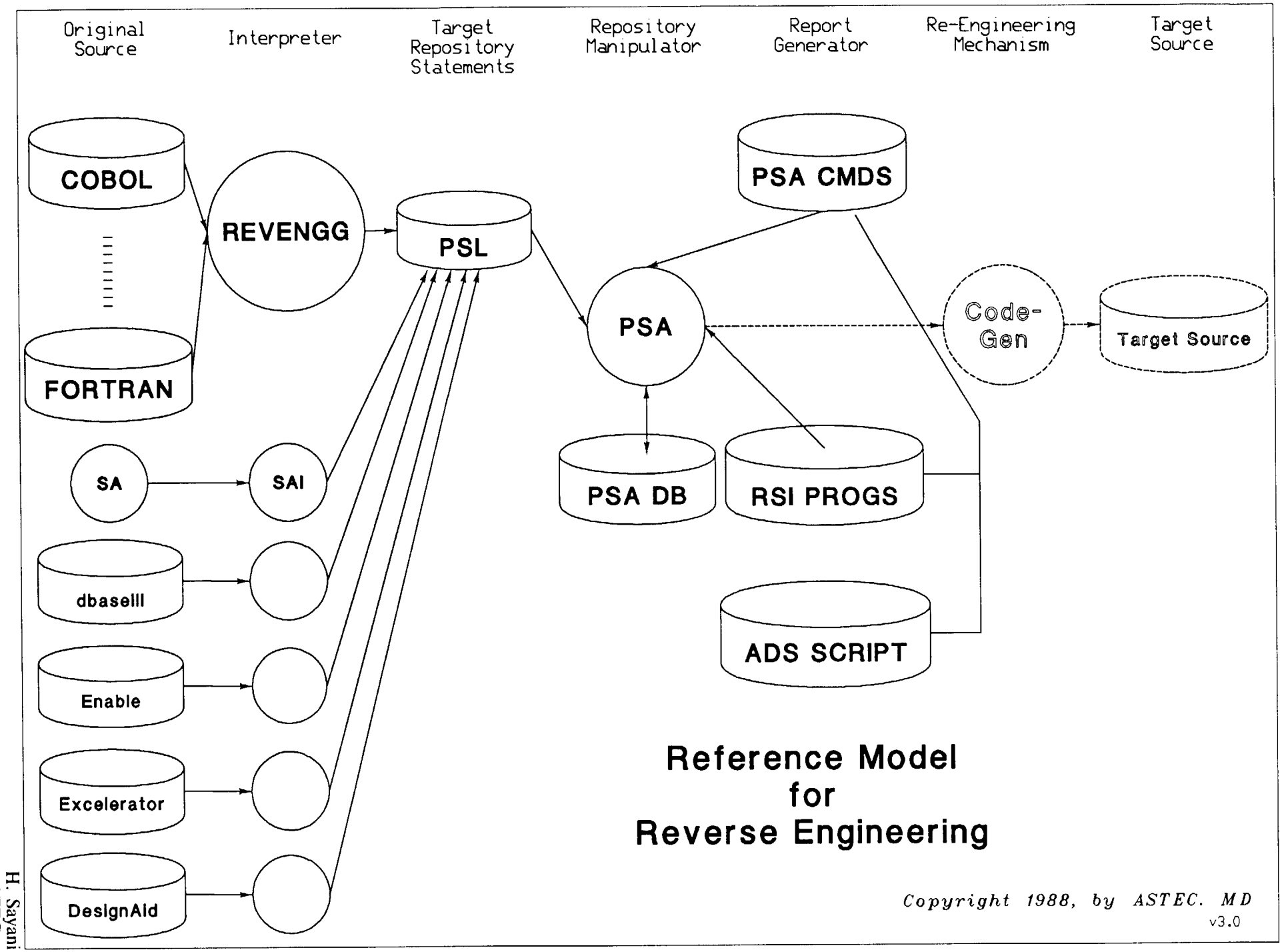
- "...Where would we best add this functionality?..."
- "...How would it affect the data structures?...."

To merge a system with another (integration)

- "...What is the common data?..."
- "...What are the new interfaces?..."

To inject new technology into an existing system

- "...Replace the various file access methods with a DBMS..."


Original Source
Interpreter
Target Repository Statements
Repository Manipulator
Report Generator
Re-Engineering Mechanism
Target Source
COBOL
FORTRAN
REVENGG
PSL
PSA CMDS
PSA
Code-Gen
Target Source
SA
SAI
dbaseIII
Enable
Excelerator
DesignAid
PSA DB
RSI PROGS
ADS SCRIPT
Reference Model for Reverse Engineering
Copyright 1988, by ASTEC. MD
v3.0

REVERSE ENGINEERING

METHODOLOGY

Steps involved:

- **Ensure that parser recognizes deviations from norm**
- **Instruct translator to handle both normal and special constructs**
- **Produce appropriate abstractions in target dictionary language**
- **Derive appropriate reports from the dictionary**
  - **For browsing**
  - **To produce formal documentation (per specific standard)**
- **Interface with other tools (e.g., a CASE Graphics package)**
- **Merge other information**
  - **About changes**
  - **Another system**

(• **Interface with other translators to reproduce code**
  - **In the same language**
  - **In another language)**

REVERSE ENGINEERING

SCENARIO OF USAGE

Analyze individual "code units" (e.g., Programs)

- Examine the Procedural architecture
- Study the Data Structure

Synthesize desired aspects across the code units of the system

- Procedural Interactions
- Data Commonality

Note: May need to rationalize names

Pinpoint answers to questions that precipitated the process

Take necessary action

- Modify the abstractions (& regenerate the code)
- Change the code

REVERSE ENGINEERING

EXAMPLE OF USAGE

Large System to be maintained:

- **A subsystem with:**
  - **45 major functions,**
    329 sub-functions,
    2711 primitive functions
  - **36 data stores**
    4,498 record types: several with over 180 elements
  - **Interactions**
    e.g., a Function uses 64 elements and changes 61
    an Element used by 422 Functions and changed by 455

Reported Savings (not counting outliers)

- **8 to 1 savings in time**
- **vastly reduced "re-work" (no unaccounted ripple effects)**

REVERSE ENGINEERING

POTENTIAL PITFALLS

From the mundane....

- Inability to transfer source code from operational environment to the Software Engineering Workbench

Through the expected...

- "...Did not know you could do THAT in FORTRAN!..."

And People's Style...

- "Why would you clump those actions in one Paragraph?..."

Along with Organizational Standards (or lack of them)...

- "That's only the third way to spell EMPLOYEE-NUMBER..."

And esoteric issues...

- "...We should get a better abstract model common to ..."

To Politics

- "...If we perform this job 8 times faster, we get paid less..."

And Technology Transfer

- N I H

REVERSE ENGINEERING

FUTURE DIRECTIONS

Improvements in the Technology

- Interpretation of Source Code
  - Broader in scope
  - Adaptive (to style and usage)
- Abstractions produced
  - Sophisticated conceptual model: across languages
- Better interfaces to CASE tools
  - Formal adaptation of Methodologies for Design and Analysis
  - Improvement in the Forward process (Traceability, Standards)
- Re-generation of program code
  - Original language
  - Different language (using "filters")

Improvements in the Technology Transfer

- The delivery platform
- Education of Engineers
- Acceptance by Management as part of the forward life cycle

Ada Software Productivity in Prototypes: A Case Study

Jairus M. Hihn
Hamid Habib-agahi
Shan Malhotra

Jet Propulsion Laboratory
California Institute of Technology

ABSTRACT

This paper is a case study of the impact of Ada on a Command and Control project completed at the Jet Propulsion Laboratory (JPL). The data for this study was collected as part of a general survey of software costs and productivity at JPL and other NASA sites.

The task analyzed is a successful example of the use of rapid prototyping as applied to command and control for the US Air Force and provides the US Air Force Military Airlift Command with the ability to track aircraft, air crews and payloads worldwide. The task consists of a replicated database at several globally distributed sites. The local databases at each site can be updated within seconds after changes are entered at any one site. The system must be able to handle up to 400,000 activities per day. There are currently seven sites, each with a local area network of computers and a variety of user displays; the local area networks are tied together into a single wide area network.

Using data obtained for eight modules, totaling approximately 500,000 source lines of code, we analyze the differences in productivities between subtasks. Factors considered are percentage of Ada used in coding, years of programmer experience, and the use of Ada tools and modern programming practices.

The principle findings are the following. Productivity is very sensitive to programmer experience. The use of Ada software tools and the use of modern programming practices are important; without such use Ada is just a large complex language which can cause productivity to decrease. The impact of Ada on development effort phases is consistent with earlier reports at the project level but not at the module level.

Introduction

The Economics Group at JPL has been involved in the collection and analysis of software cost and productivity data for the past three years. The NASA Historical Database contains data for over 100 subsystems including 10 different projects. [Economics Group 1989] The JPL Software Database currently contains data for 4 projects with 39 subsystems.[SORCE/Economics Group 1988] During the coming year data on seven more projects will be collected. A relatively unique feature of these databases is that they contain data on all the subsystems of each project for which information could be obtained. Most software databases used for research contain only one or two observations from any one project. The advantage is that we are able to control for differences between projects which are not directly measured by the specific database fields and also can also analyze within project variations in effort and productivity. The disadvan-

tage is that a larger number of observations must be collected to get a sufficient number of independent data points for statistical analysis.

The data collected is primarily based on the COCOMO definition of a software environment. [Boehm, B. 1981] Table 1 lists the cost driver contained in the database which describe the environment. The database also includes size, measured by executable source lines of code adjusted for inherited and modified code, and effort, measured by work months. The portion of the life cycle for which effort figures have been collected includes from the requirements analysis phase through test and integration. Sustaining engineering and the systems engineering effort to develop the requirements are nor included. However systems engineering effort spent on requirements design updates and formal design reviews is included. Two estimates of effort were collected. Technical effort figures gathered from interviews with the technical leads, estimates direct effort by programmers and the technical managers. Implementation effort figures derived from the task management office, include all labor charges to the project from the task manager down. The non-direct labor charges are distributed across the subsystems on a proportional basis. These charges include integration and validation testing, documentation and management labor time. Implementation effort also includes secretarial time which could not be separated out. Effort figures do not include upper level project management or system engineering previous to the SRR.

Table 1
Database Description

Product Attributes	**Computer Attributes**
Required reliability	Time constraints
Software complexity	Storage constraints
Database size	Host volatility
	Turnaround time
Personnel Attributes	**Project Attributes**
Analyst ability	Software tools
Analyst experience	Modern programming practices
Programmer ability	Schedule
Language Experience	
Virtual Experience	

The average productivities in the NASA Historical Database are 1.5 to 3.5 SLOC per day for flight software and 7 to 10 SLOC per day for ground based software. There were a few subsystems which reached approximately 14 SLOC. In the JPL Software Database, the average productivity ranged from 6 to 18 for 3 DOD projects and one ground data capture project. There are two command and control projects which had the highest productivities of the projects we have studied. Project 1 used Ada and rapid prototyping to reach a implementation productivity of 17.9 SLOC/ work day. Project 2

which was very similar to Project 1 did not use Ada and had an implementation productivity of 13.5 SLOC/ work day. The purpose of this study is to attempt to isolate the impact of Ada versus the impact of software tools, modern programming practices and other environmental factors on productivity.

Project Description

The US Air Force Military Airlift Command (MAC) runs one of the largest airlines in the world. Scheduling problems are accentuated because flights, crews, and payloads can be changed at any time in order to meet political and military objectives. MAC is in the process of automating its command and control system by replacing its current scheduling system, based on grease boards and the telephone, with a network of workstations supporting a replicated database with real-time displays. Two major components of MAC's Command and Control Automation Project are being completed by JPL. Project 1 supports the vertical command and control operations, and Project 2 supports the actual execution of tasks. Project 1, a successful example of the use of rapid prototyping, consists of a globally distributed replicated database with sites from Germany to Hawaii.

Developed as a prototype which became an operational system, Project 1 had an unusual software life cycle for a delivered system. JPL was required to develop Project 1 within two years at minimal cost. The functional requirements were vague because the sponsor was not very computer literate. The project manager compensated for these factors by waiving many of the standard formal design, documentation, and testing requirements and by developing a very close working relationship with the sponsor. The final requirements evolved as part of a joint effort between the project team and the sponsor. Detailed documentation, except for the user's guides, could be written after the project team and the sponsor had agreed that the system was working.

Project 1 consists of five application subsystems and three support system subsystems. The applications support the following five MAC functional groups: Current Operations (DOO), Transportation (TR), Command and Control (DOC), Logistics (LRC), and the Crisis Action Team (CAT). The software work breakdown structure is similar to the functional breakdown; therefore, the descriptions which follow of functional groups also serve as descriptions of corresponding software tasks. DOO performs flight scheduling and resource planning. TR is responsible for personnel ticketing and cargo loading and unloading. DOC monitors the progress of each flight. When en route mechanical failures occur, LRC provides information which assists in the prompt servicing of debilitated aircraft. CAT controls system responses in the event of a threat or emergency.

System support for Project 1 resides in three subsystems: Graphics, Operating System Shell, and Database. Graphics produces a graphical display of database information while allowing the user to manipulate screens via a user interface. Operating System Shell provides an interface to VMS OS, network commands, and low level VMS functions. Database supports database design and control.

Table 2

Development Data

Subtask	Size (KSLOC)	Technical Effort (Work Months)	Implementation Effort (Work Months)	Technical Productivity (SLOC/day)	Implementation Productivity (SLOC/day)
Application Software					
DOC	72	118	207	32	18
DOO	115	140	245	43	25
LRC	45	28	49	84	48
CAT	23	36	63	34	20
TR	70	60	105	61	35
System Software					
Graphics	20	72	145	15	8.3
Common	110	258	453	22	13
Database	37	110	193	17.7	11
Total	492	822	1,460	31.5	17.9

The development data collected was based upon the status of the project in January 1988 which was before the software system was actually converted into a formal product. The total size was approximately 500,000 source lines of code.[1] The sizes of the modules range from 20,000 to 115,000 source lines of code. The code count is based on executable source lines of code; the size figures do not include comments or blank lines.

The productivity figures for the Project 1 subsystems are presented in Table 2. The average technical productivity of Project 1 as a whole was 31.5 source lines of code per day; the average total productivity was 17.9 source lines of code per day. At the time of final delivery of the system implementation productivity had increased to an average of 20 SLOC/ work day. This occurred even though documentation and testing effort increased significantly during the last release. This is most likely a result of the staff being further up on the learning curve with respect to Ada and the application domain. Among

1. At final delivery Project 1 will have reached approximately 750,000 source lines of code.

the systems tasks, total productivities averaged 11 and ranged from 8 to 13 source lines of code per day. The application tasks had total productivities averaging 25 and ranging from 18 to 48 source lines of code per day. In general, application software is associated with higher productivities than system software because application software is less embedded and usually does not have to incorporate low level implementation details.

Table 3 summarizes the values of the environmental factors included in the database for Project 1. However, the table shows that experience and capability were rated high; requirements volatility was rated low; and the use of modern programming practices and software tools was extensive throughout the project.

Table 3
Project 1
Development Environment

Product Attributes	Low to Nominal
Computer Attributes	Low to Nominal
Personnel Attributes	High
Project Attributes	High

ANALYSIS

Project 1 developers achieved higher total productivity than the average NASA project teams developing ground software. Several factors combined to permit this achievement: the ability to match highly qualified personnel to the task needs, the use of a prototyping methodology, the organizational structure of the development team, an abundance of development tools, excellent communications with the sponsor, development team cohesiveness, and the use of Ada.

The development environment contributed to the high productivity of the project staff. The implementation managers were able to match skills and project needs with programmers whose capability and experience were well above average. Project 1 was developed as an incremental prototype; the development strategy cut the standard development life cycle. User's guides were written in parallel with the software. A single design document was written at the end of the project which was the equivalent of an FRD, FDD, SRD, and SDD combination to assist during the sustaining engineering phase. In the testing phase a formal independent validation and verification was omitted; and there were no formal preliminary and critical design reviews by an external organization. However, there was a formal internal review prior to each major software re-

lease. The small overall staff size facilitated open communication within groups, between groups, and with the sponsor. The sponsor provided ample hardware which was appropriate to each task. Finally, the majority of the tasks were of moderate difficulty or complexity.

One other factor that potentially contributed to the high productivity of Project 1 was the use of Ada. At the time of the initial survey fifty percent of the total code was Ada and varied from 0 to 90% across the subsystems. When the project started about half of the programmers had an average of 1 year experience with Ada and the rest had no experience. A few had the maximum possible experience of about 2 to 2.5 years. There was no formal requirement that Ada had to be used. In the early 1990's Ada will be a more mature language, but this level of staff quality was the best that could be hoped for when software development began two years ago.

Ada advocates claim that the proper use of Ada, with its software tools, strong type checking, and support of modern programming practices, increases programmer productivity by over 100% and decreases program maintenance costs[Royce, W. 1987].

It is difficult to test these claims, however, because one must be very careful when comparing the productivities of programmers coding in different languages. In particular, Ada has several characteristics which can cause an Ada program to have more or fewer lines of code than other third-generation language programs with the same functionality. Ada's syntax for using objects can inflate an Ada program's code count. On the other hand, Ada's ability to use generic procedures can deflate Ada's code count, since a generic procedure would have to be written a number of times in a third generation language. A recent survey found that the effect of Ada on code count depends upon the application: business and scientific applications tend to result in larger Ada code counts whereas avionics and automation projects tend to have smaller Ada code counts.[Reifer, D. 1988]

Accurate measurement of the impact of Ada on productivity requires that major differences between organizational structures also be isolated. When subsystems of very different projects are compared environmental differences not captured in the data can arise. These differences especially relate to environmental factors such as communication between sponsor and contractor and cohesiveness of the programming teams. The result is very large variances in the data; conceptually the problem is that of comparing 'apples and oranges'.

The results of productivity comparisons between different projects and especially between languages is very sensitive to both the type of application and unexplained environmental factors. To reduce the impact of these problems we will emphasize comparisons between modules with similar amounts of Ada and comparisons between projects that are very similar in nature. The other project, Project 2, that will be referenced in the analysis is also a command and control task performed under the same project office at JPL and also for MAC. Both tasks were eventually housed in the same building and both were prototypes at the time of this survey.

Figure 1
Implementation Productivity Unadjusted

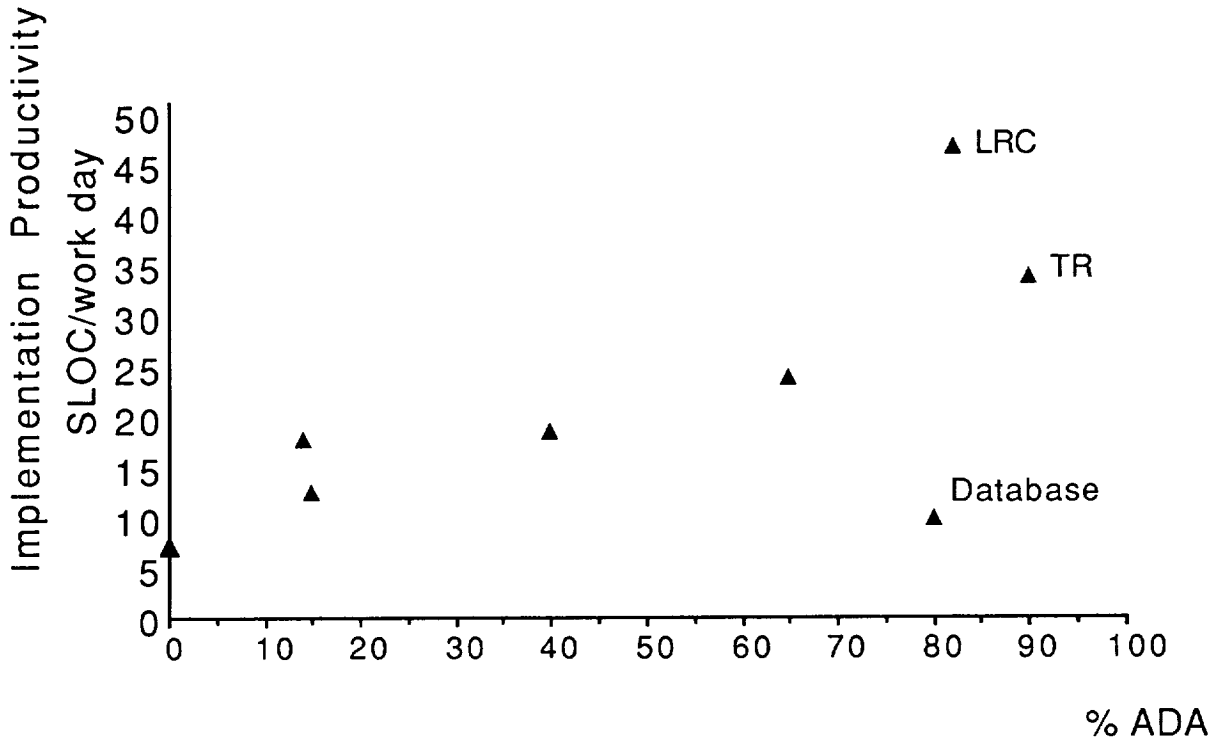


Figure 1 plots productivity, (SLOC/implementation effort)/19, against %Ada, the percent of code in Ada for a module.[2] The graph is suggestive of a positive correlation between the percent of a subsystem's code written in Ada and the productivity of that subsystem which would represent the combination of the impact of Ada and the cost of mixing languages. Comparing the average productivity of those modules with less than 50% Ada to those with greater than 50% Ada one may be tempted to draw the conclusion that use of Ada increases productivity by about 15 SLOC/day which would be close to a 100% increase. However there are many other differences between these subsystems which also impact productivity and these must be identified in order to isolate the actual impact of Ada on productivity.

For example, compare the productivities of subtasks with similar percents of code written in Ada. LRC, TR and Database are three such modules. Programmer experience and the use of modern programming practices and tools are significant differences between these subsystems. At the time of the survey the LRC technical lead, which achieved the highest productivity, had 2.5 years of experience coding in Ada and six years of experience object-oriented design. The nature of the LRC task allowed the team to use objects extensively. The LRC staff also consistently employed modern programming practices and software tools. The productivity of the TR team

2. 19 represents the actual number of work days in a month when discounting for holidays, sick days and general meetings. [Boehm, B. 1981]

was lower than that achieved by the LRC staff; the TR staff did use modern programming practices and tools, but the TR programmers, with one to two years of experience coding in Ada, were less experienced than the LRC team members. The Database team were less productive than either the LRC or TR teams. Database had zero years Ada experience because the only Database Ada programmer left the project on very short notice. The remaining team members were left to tackle a complex task with high required reliability while learning to use a complex language. The inexperienced Ada team did not use software tools and did not follow modern programming practices. However, the following question remains: just how much of the productivity differences do experience, tools and modern programming practices when combined with Ada explain?

Before we can answer that question we need to control for other known environmental influences. Some projects are more complex; others have a greater required reliability. If the database were large enough, we could estimate the influence of the environmental factors including the presence of Ada. Since there is not sufficient data, a second best solution is to use known estimates of the effort impact of the environmental factors. COCOMO provides estimates based on non-Ada projects. Therefore we can normalize for these factors using the COCOMO weights, and the remaining productivity variations between modules are likely to be related to the presence of Ada.

Assuming that

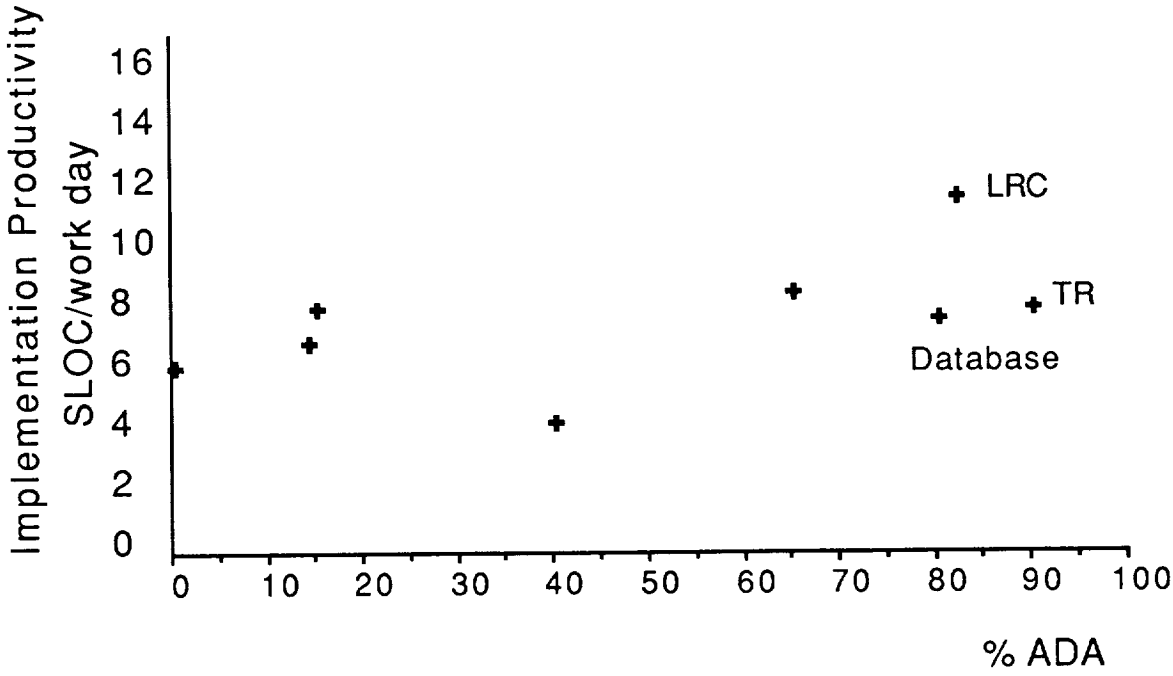
$$\text{Effort} = A \cdot L^{B} \cdot \text{EAF}$$

where L is executable source lines of code and EAF is the product of the cost drivers or environmental factors then adjusted effort is just Effort/EAF. Adjusted productivity then becomes

$$\text{ATOP} = [L/\text{Effort}] \cdot \text{EAF}.$$

Figure 2 displays the plot of adjusted productivity against % Ada . After adjusting for all the software development environmental factors except language experience the adjusted productivity values vary from 6.1 to 11.7 SLOC/work day. All but two subsystem adjusted productivities fall between 6.1 and 8.6 SLOC/work day.

Figure 2
Productivity Adjusted
Except for Language Experience



The average productivity for those module with less then 25% Ada is 6.9 SLOC/work day and for those modules with greater then 60% Ada it is 9.1 SLOC/work day. Based on a two-tailed t-test there is only a 10% probability that these represent the same distribution. Hence we can tentatively conclude that those projects with a high Ada content had a productivity 2.2 SLOC/work day higher then those with little or no Ada. Compared to the average productivity for the whole project this represents a 12% increase.

Within the group of modules with greater then 60% Ada the LRC module attained the highest productivity of 11.7 SLOC/work day which represents a 4.8 SLOC/work day increase or 25% improvement. The high productivity of Logistics is probably reflective of their being further up on the learning curve. Logistics did have one member who had the maximum possible Ada experience and substantial experience with object oriented programming. This suggests that three years of experience with Ada and an Ada programming environment might represent an important turning point. This point is further reinforced given that during the final release productivity increased to 27 SLOC/work

day which is when those who started with about 1 year of Ada experience would have reached over three years of experience.

One other comparison that can be made is to compare the adjusted productivities between two similar projects one which uses Ada and one that does not use Ada. The comparison project used Pascal. These results are reported in Table 3. The comparison project is also a command and control task for the Air Force and even for the same contractor. The one major difference that cannot be controlled for is that Project 1 started out as a prototype but became an incrementally developed delivered system and project 2 was a prototype from beginning to end. After adjusting for differences in complexity and the lack of software tools the non-Ada project has a higher average adjusted productivity. Based on a two-tailed t-test there is only a 5% probability that these represent the same distribution.

The implication is that if you take away the tools and rules and adjust for differences in complexity and other environmental factors then the main impact of Ada as a language, without its tools lowers productivity when the programming staff has an average of one year experience. From the previous discussion we also suspect that once the experience level gets above three years then this difference will no longer be statistically significant.

Table 3

Average Productivity
(SLOC/work day)

	Total	Adjusted
Project 1 (Ada & C)	17.9	7.7
Project 2 (Pascal)	13.6	13.6

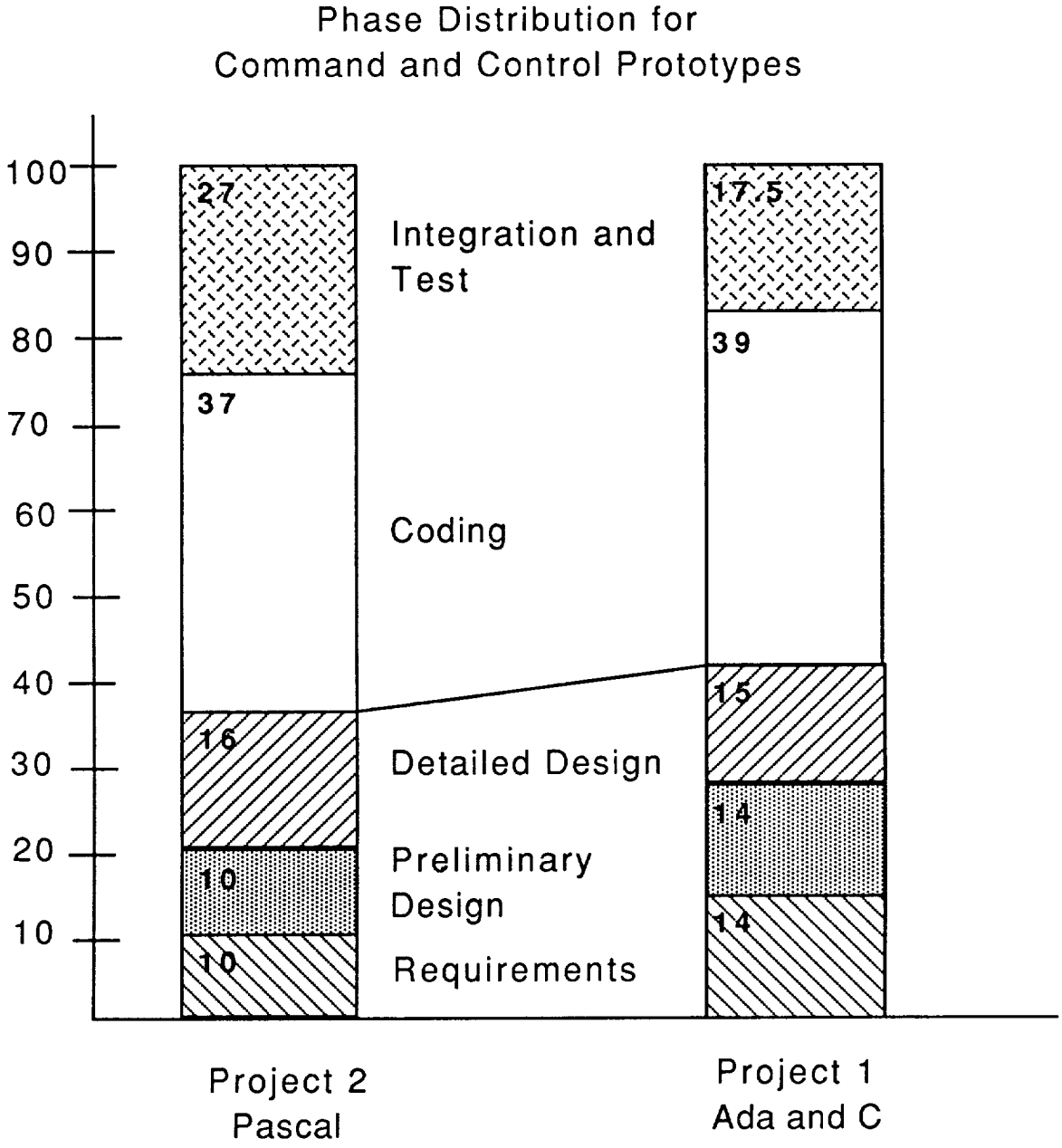
For this small sample the inference that can be drawn is that for experience of one year or less we can explain the majority of the observed variation in productivities by what we know about the impact of software tools, experience, etc on other languages. Software tools are important and a sophisticated programming environment will increase the productivity of any language. This interpretation must be discounted by the fact that Project 1 is a prototype and therefore the testing and integration phase plays a less significant role in determining development costs and it is here that one would expect Ada to have its most significant impact on development effort and productivity.

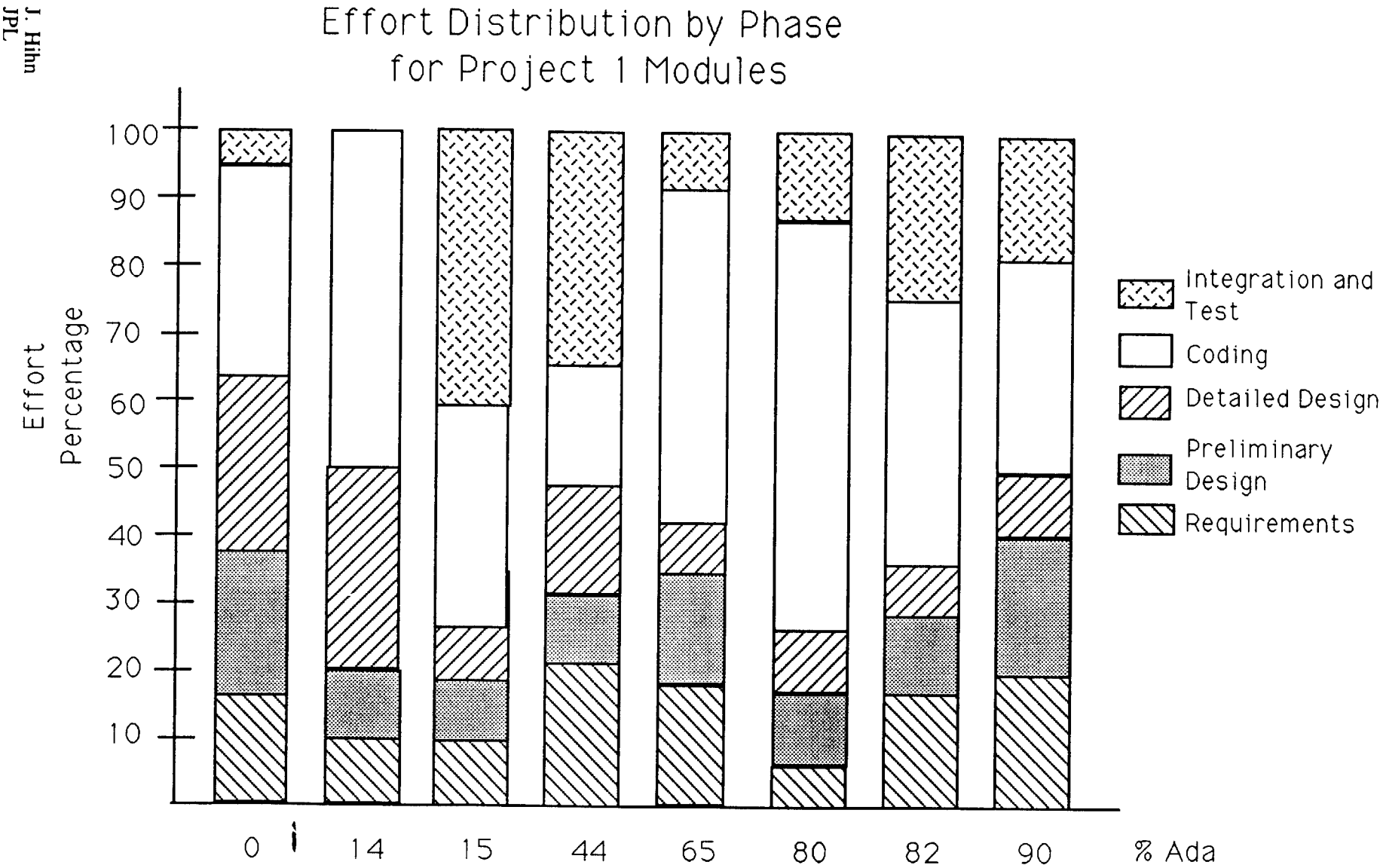
Ada and the Development Life Cycle

Previous studies have reported that Ada increases the effort in design, and decreases

effort in the integration and test phase. One phase breakdown that has been reported is 50:33:17 for Ada and 40:38:22 for FORTRAN.[Royce, W. 1987] Comparing to Projects 1 and 2 again we can see to what extent this pattern holds up for prototypes. Figure 3 shows a phase breakdown for the whole project of 36:37:27 for Pascal and 43:39:18 for an Ada and C project. As expected, prototypes spend less time in design and more in coding. Furthermore the Ada prototype spends more time in design and less in testing then the non-Ada prototype.

While the effort by phase breakdown for the projects as a whole yields a consistent story the view from the module level does not. There does not appear to be any consistent pattern whatsoever.




Effort Distribution by Phase
for Project 1 Modules
Effort
Percentage
100
90
80
70
60
50
40
30
20
10
0
14
15
44
65
80
82
90
% Ada
Integration and
Test
Coding
Detailed Design
Preliminary
Design
Requirements

Conclusion

The data reflects the state of Project 1 before it actually became productized and therefore contains reduced effort figures for testing and documentation, which greatly increased during the final release. There is also not any data on maintenance costs. Therefore the two areas where Ada's strong type checking and compiler have an effect are not reflected. In addition there was no effort to make the code portable or reusable.

Any conclusions are tentative and should be treated as hypothesis for future research. As part of our continuing software costing analysis at JPL, two Ada projects and one Lisp project will be surveyed during 1989 which should make it possible to better isolate the impact of software tools and modern programming practices from other features of a language.

Given these caveats then our tentative conclusions are the following for Ada in a prototyping environment.

(1) Analyst and programmer experience in Ada of three years or more could increase Total Technical Productivity by 3-4 SLOC/day or a maximum of 25%.

(2) Technical experience and ability, modern programming practices and the use of software tools are very important in achieving high programmer productivity.

(3) For any language the combination of highly capable and experienced personnel, with the discipline of modern programming practices and a sophisticated programming environment should produce comparable levels of productivity to that observed for Ada in this study.

(4) Effort in the three major phases of the software lifecycle appears to shift such that time spent in design is increased and time spent in verification and test is decreased.

Bibliography

Boehm, B. 1981 Software Engineering Economics, Prentice Hall.

Economics Group 1989 NASA Historical Database, JPL/Caltech, January 1989.

Reifer, D. 1988 Softcost-Ada: An Update, Fourth Annual COCOMO Users' Group Meeting Workshop, Pittsburgh, Pa., Nov. 2-3, 1988

Royce, W. 1987 Estimating Ada Software Development Costs for C^3 Systems, TRW Defense Systems Group, Preprint.

SORCE/Economics Group 1988 Software Productivity Analysis Database, JPL/Caltech, 1988.

THE VIEWGRAPH MATERIALS

FOR THE

J. HIHN PRESENTATION FOLLOW

Ada Productivity Analysis: A Case Study

Jairus M. Hihn
Hamid Habib-agahi
Shan Malhotra

Jet Propulsion Laboratory
California Institute of Technology
December 13, 1988

Outline

Project Overview

Command and Control Projects

Ada Claims

Analysis

Conclusions

Software Database Description

- *Size* - executable source lines of code (SLOC) without comments
  adjusted for inherited and modified code
- *Cost* - work months assuming 19 working days per month
  technical effort from interviews and implementation effort from task management
- *Effort breakdown by phase*
- *Development mode*
- **Product Attributes**
  Required Reliability
  Software Complexity
  Database Size
  Development Language
- **Computer Attributes**
  *Time Constraints**
  *Storage Constraints**
  *Virtual Machine Volatility**
  Turnaround Time
- **Personnel Attributes**
  Analyst Capability
  Analyst Experience
  Programmer Capability
  Language Experience
  Virtual Machine Experience
- **Project Attributes**
  Software Tools
  Modern Programming Practices
  *Schedule**
  Requirements Volatility

Average Productivities

NASA Historical Database, 10 projects, 100 modules

Flight S/W averaged approximately 1.5-3.5 SLOC/work day

Ground S/W averaged approximately 7-10 SLOC/work day

JPL Software Database, 4 projects, 39 modules

Project	SLOC/work day
1	17.9
2	13.6
3	15.4
4	6.5

Project 1
Command and Control

Development Data

Module	Size (KSLOC)	Technical Effort (Work Months)	Implementation Effort (Work Months)	Technical Productivity (SLOC/work day)	Implementation Productivity (SLOC/work day)
Application Software					
DOC	72	118	207	32	18
DOO	115	140	245	43	25
LRC	45	28	49	84	48
CAT	23	36	63	34	20
TR	70	60	105	61	35
System Software					
Graphics	20	72	145	15	8.3
Common	110	258	453	22	13
Database	37	110	193	17.7	11
Total	492	822	1,460	31.5	17.9

Project 1
Development Environment

Ada

Prototype

- Documentation
- Testing

Project combined known technologies and techniques

Excellent communication existed between users and developers

Personnel were well matched to tasks

Project 1
Development Environment

Product Attributes	Low to Nominal
Computer Attributes	Low to Nominal
Personnel Attributes	High
Project Attributes	High

Ada Claims

Ada will lower life cycle costs in long term projects

- Maintenance costs

Ada will/may lower development costs

- Integration and test
- Coding

Caveats

The results should be viewed as hypotheses for future research since

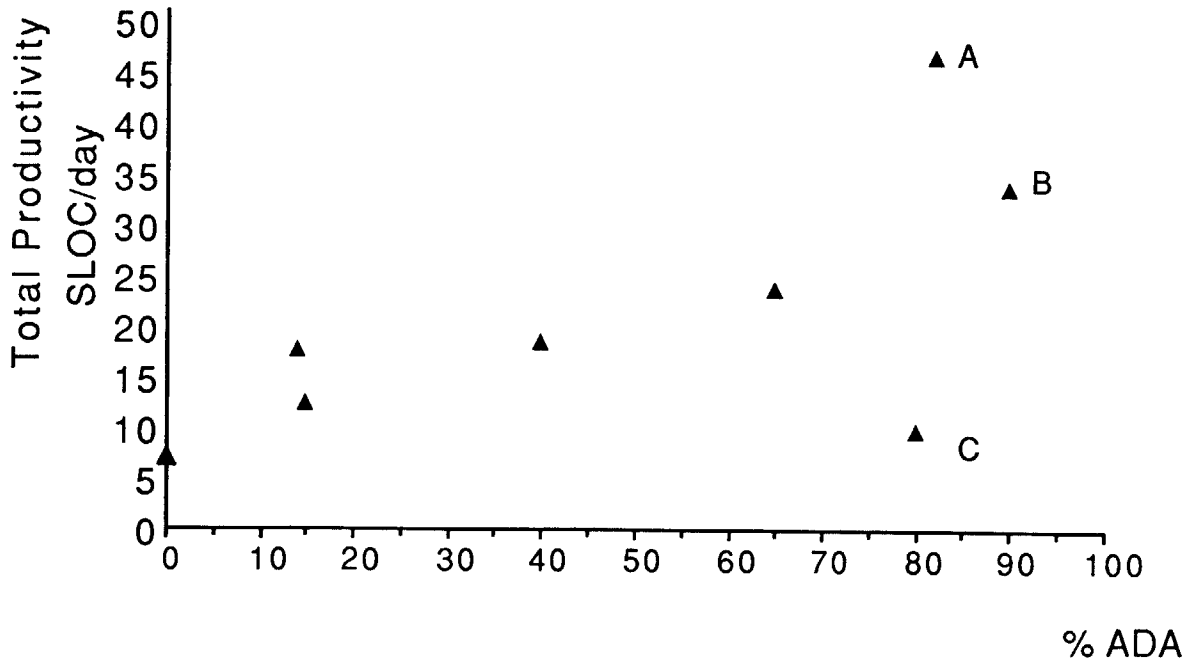
the sample size is small

care must be taken when comparing "lines of code" from different languages

effort was not expended to support reuse or portability

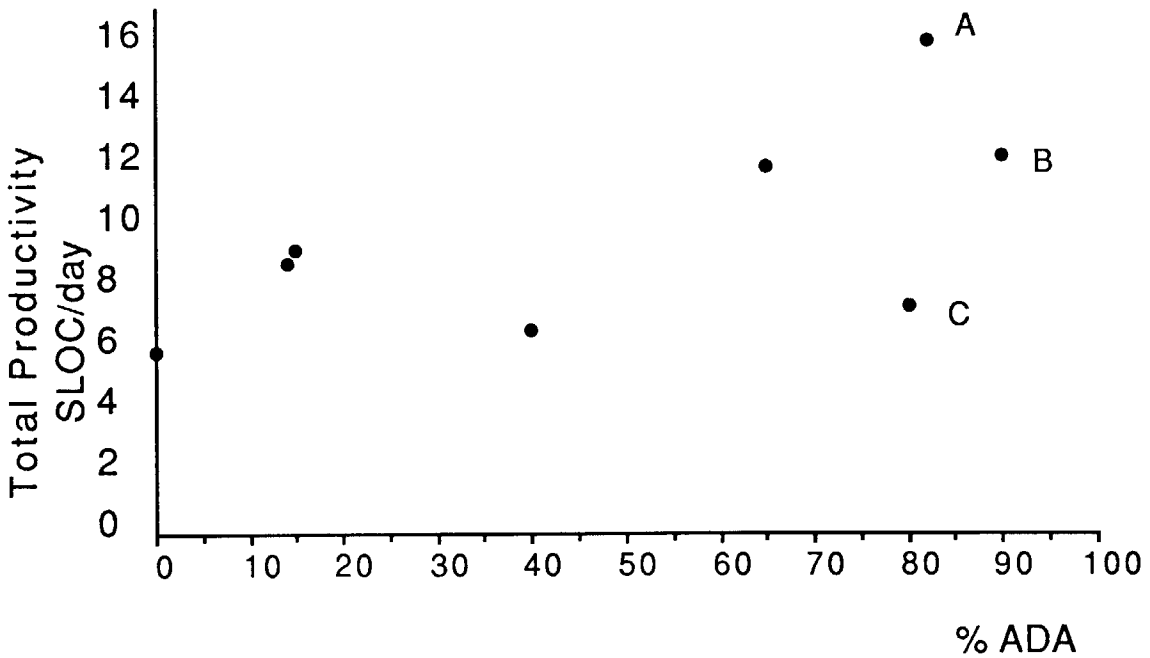
the data represents a prototype and Ada is designed for production systems

Impact of Ada on Productivity ?

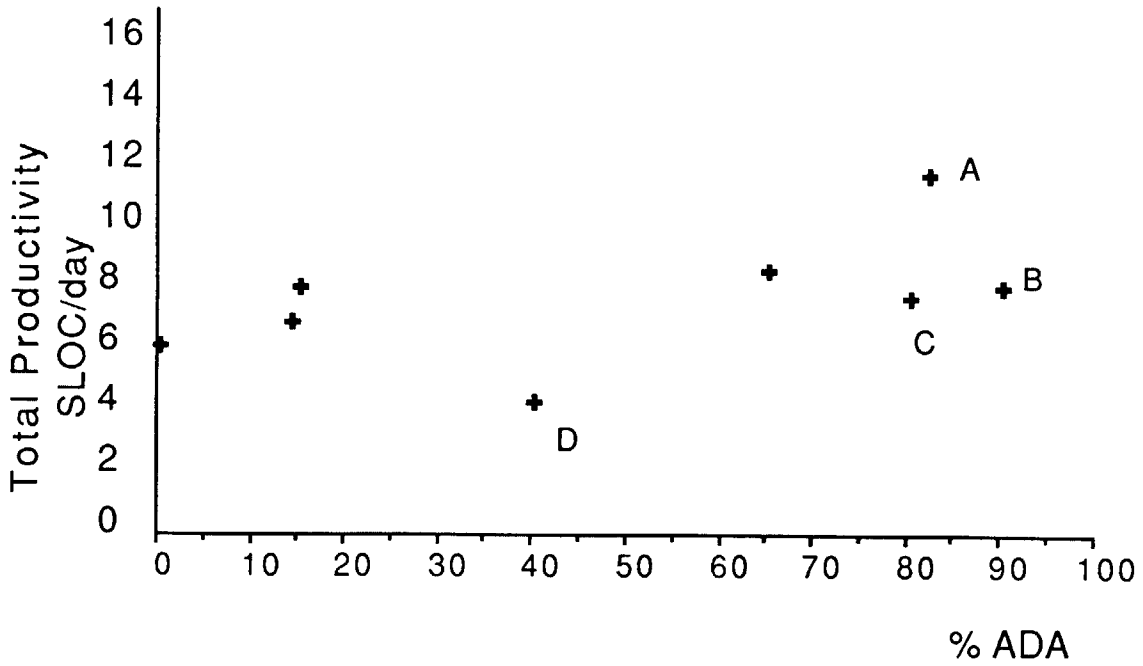


There appears to be a general increase in productivity as the % of Ada increases in a module.

Productivity Partially Adjusted
Except for Software Tools, Modern Programing Practices and Language Experience



Productivity Adjusted
Except for Language Experience



Project 1 compared to Project 2
Command and Control

The average productivity for project 1 (Ada and C)

Total productivity = 17.9 SLOC/work day

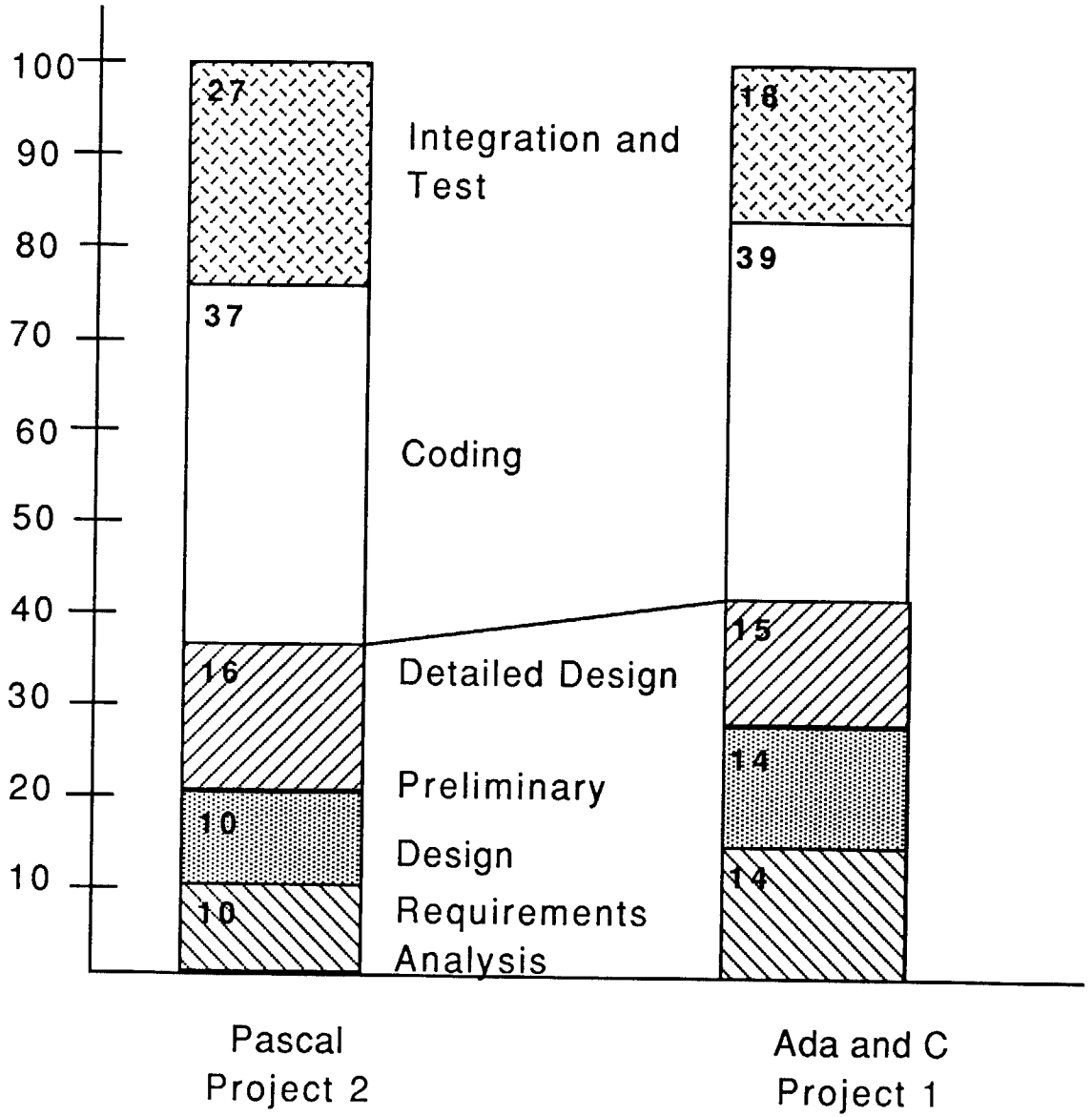
Adjusted productivity = 7.4 SLOC/work day

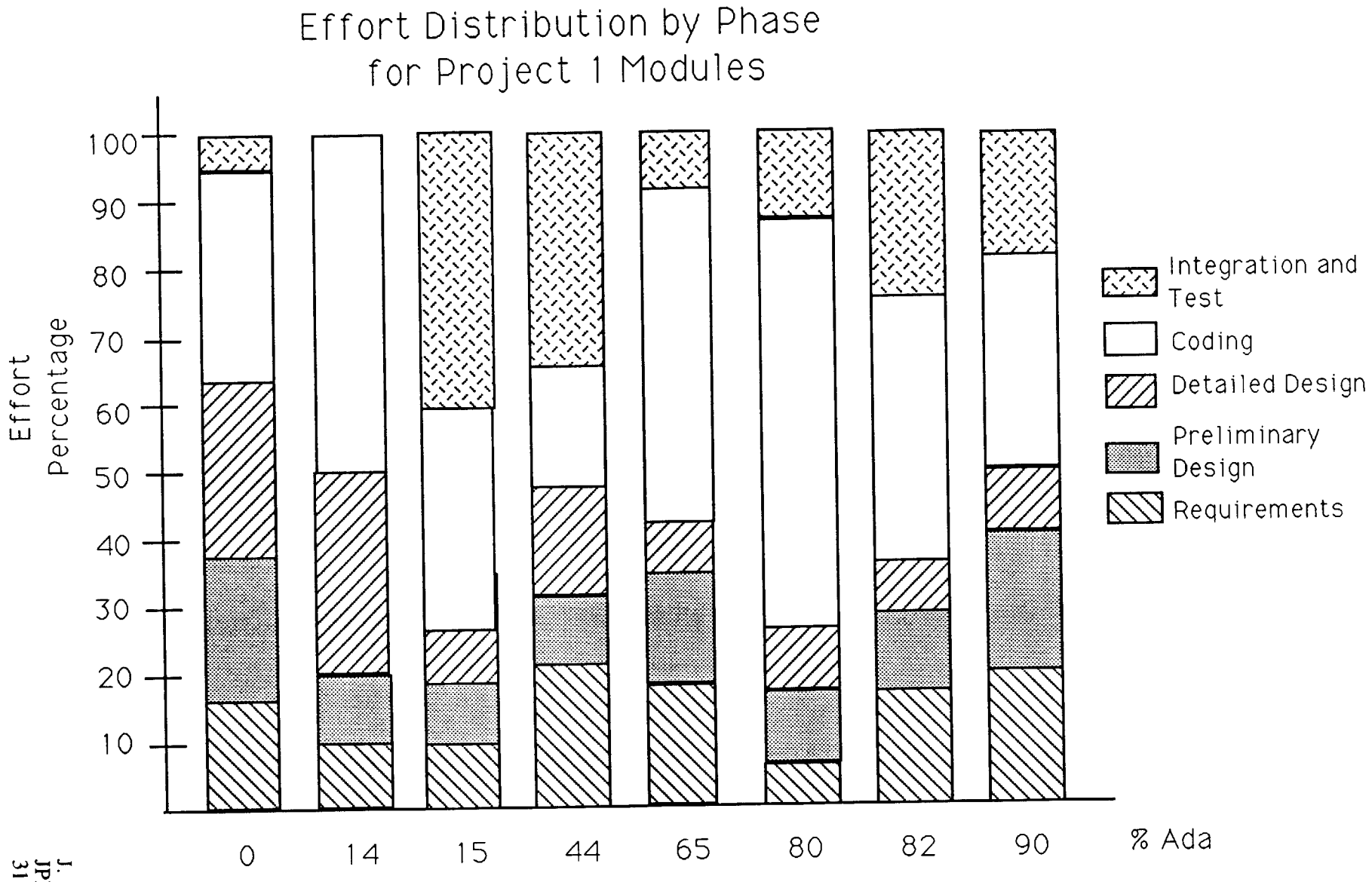
The average productivity for project 2 (Pascal)

Total productivity = 13.6 SLOC/work day

Adjusted productivity = 13.5 SLOC/work day

Phase Distribution for Command and Control Prototypes
Project Level




Effort Distribution by Phase
for Project 1 Modules
Effort
Percentage
100
90
80
70
60
50
40
30
20
10
0
14
15
44
65
80
82
90
% Ada
Integration and Test
Coding
Detailed Design
Preliminary Design
Requirements

Conclusions

These results are reflective of a new project which has hired very capable people with extensive general experience but on average 1 year experience in Ada.

10-25% increase possible with Language experience > 3 years

Similar productivity gains should be possible with other languages

- Programming environments
- Modern programming practices

Impact of development phases

- Increased design and requirements
- Decreased integration and test

Experiences with Ada in an Embedded System

Robert J. LaBaugh
Martin Marietta Astronautics Group
Space Systems
Denver, Colorado 80201

Introduction

This paper describes recent experiences with using Ada in a real time environment. The application was the control system for an experimental robotic arm. The objectives of the effort were to experiment with developing embedded applications in Ada — evaluating the suitability of the language for the application, and determining the performance of the system. Additional objectives were to develop a control system based on the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) in Ada, and to experiment with the control laws and how to incorporate them into the NASREM architecture.

Background

The arm to be controlled has five degrees of freedom — one degree in each of the shoulder and elbow joints, and a wrist with roll, pitch, and yaw. An Intel 80386 single board computer in a Multibus II system was used for the controller. The board contained an 80387 math coprocessor, two megabytes of RAM, and a single RS-232 serial port. The clock frequency for the system was 16 MHz. Rather than just use the 80386 as a fast 8086, the 80386 was operated as a 32 bit processor in the protected mode, which provides for segment sizes of up to four gigabytes.

The Ada compiler selected was the DDC-I cross compiler for the 80386, which was hosted on a MicroVAX. This compiler was targeted to a bare machine, so there was no operating system to either provide services or detract from the performance of the system. The runtime system supplied with the compiler provided all of the services needed to support the features of the language, including initialization of the hardware, memory management,

time management, the Ada tasking model, and interrupt handlers. An operator interface for the application was implemented using the standard Ada Text_IO package. This package uses the RS-232 port on the single board computer for the standard input and output of Text_IO.

Development Approach

The software development system is shown in Figure 1. It consisted of a Rational R1000, a MicroVAX II, and a PC clone. The systems were connected via Ethernet, which was used to transfer files between the systems. Initial program development was done on the Rational. To facilitate code debug and checkout on the Rational, Ada routines to simulate the hardware were developed. These were used to replace the low level hardware interface routines. When the target hardware and compiler became available the source code was moved to the MicroVAX. Target peculiar modifications were made to the code, such as the specification of task entries as interrupt handlers and the hardware interface routines. The code was then compiled and linked on the MicroVAX, and the resulting load module was downloaded to the PC. The PC served as the controller for the in-circuit emulator, which was used to load and control the execution of the code in the target system.

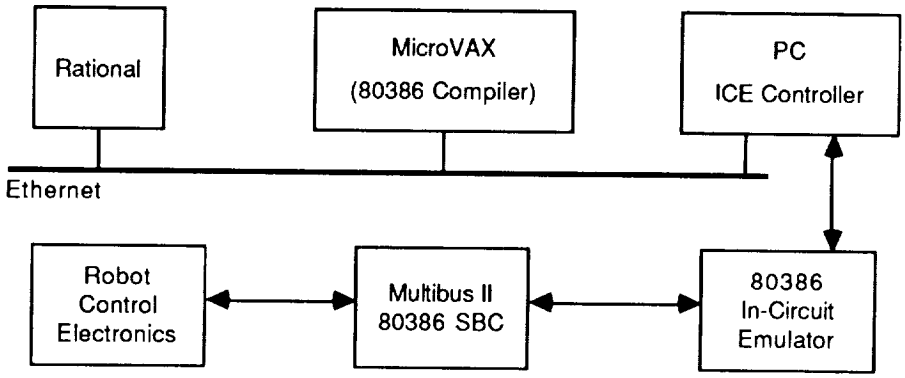


Figure 1. Development System

Even though the capabilities of in-circuit emulators are improving, this was a less than optimal environment for debugging code. Having to move from one terminal to another, moving files from one system to another, and the limitations on file names on the PC all hinder code development and checkout. The movement is clearly toward being able to compile, download, and debug from a terminal on the host development system. There are some systems which currently allow this, but the targets are connected to the host by an RS-232 line. The relatively slow download speeds limit the size of programs which can be effectively developed using these systems.

Ada Features Used

Ada tasks and task rendezvous were used for synchronization and communication between tasks. Task priorities were established using the priority pragma. An interrupt handler was coded in Ada to service the timer used to provide the control loop cycle. This was accomplished using an address clause for a task entry — which is the technique specified in the Ada Language Reference Manual for defining interrupt handlers. The Low_Level_IO package was used to communicate with the hardware controlling the joints on the arm.

There was one package where machine code insertions were used. This was used to provide procedures to disable and enable interrupts. These routines were not really needed by the initial application. They were used to assure safe initialization of the hardware, which was already guaranteed by the sequencing of the initialization routines. However, these routines become necessary as more Multibus II features are used. This is because some logical operations, such as accessing a single Multibus II interconnect space register, require accesses to multiple hardware ports.

Software Application

NASREM defines a layered, hierarchical control system with common interfaces between layers. The lowest layer in the hierarchy operates at the highest frequency, with a decreasing frequency of operation with each higher level. Ada tasks were used to implement the NASREM layers, with the priority of the tasks decreasing with increasing levels in the hierarchy. The requirement that the argument to the priority pragma be a static expression

prevented the use of a generic package in defining the NASREM levels. However this was a minor inconvenience as there was very little code involved in defining the control structure within a level.

The initial application concentrated on the two lowest levels of the NASREM architecture. The servo level reads current joint positions and sends motor commands based on the error between the current and desired position. This level was driven by a programmable hardware clock which generated a periodic interrupt. The primitive level determines evenly spaced points between desired end points and performs the kinematic transformations. The elemental move level initially consisted of simple canned motion generators, and the task level simply selected the motion to be performed. The robot control function and the operator interface were both run on the same CPU, with a total of eleven Ada tasks in the application.

The entire application was coded in Ada. No non-standard pragmas or special interface routines to the runtime system were used. In addition, we were able to effectively write low level code in Ada. This included interrupt handlers, hardware interface routines, Multibus II message passing routines, and control of a DMA processor. The hardware, and the code generated by the compiler, provided more than adequate performance for the system. In experimenting with the control laws the control loop cycle time was varied between 10 and 50 milliseconds. For most of that range all levels of the NASREM architecture were able to complete in a single cycle. Since the NASREM architecture is set up for approximately a ten to one ratio in frequency of operation between levels, this leaves plenty of room for growth.

Current activity includes splitting the robot control function from the operator interface function and executing them on two CPUs. The initial interface and communication between the processors is via shared memory. As an alternative, Multibus II message passing will also be investigated. This is being done as an exercise in distributing the application. Items of interest are the difficulty of implementing various communication schemes and the relative performance.

Lessons Learned

Most of the things which could be considered lessons learned are more appropriately classified as common sense. Specifically, while being able to use a host system for initial debug and test is a useful development tool, it does not eliminate the need for low level testing on the target system. This testing is needed to establish the correctness of the hardware-software interface definitions, and to build confidence in both the hardware and low level software routines. Having a set of programs to incrementally checkout the low level functions and interface also provides the basis for trouble-shooting as problems arise. Such routines were needed to isolate hardware failures and identify improper system initialization, which happened if a specific sequence was not followed for powering on the electronics racks and computers.

Another major lesson learned was that portability is not automatic with Ada. There were two specific instances of this. The first involved differences in the tasking implementation between the Rational and the 80386 target. Tasks of equal priority are time sliced on the Rational, but this is not the default for the DDC-I runtime system. A task which was to run in the background, and which checked flags in an infinite loop, was elaborated before some of the higher priority tasks were initiated. Since the task didn't allow for any type of context switch, as soon as it started executing on the 80386 it kept control of the CPU, preventing the further elaboration of the system. Inserting a delay statement inside the loop fixed the problem. The other experience with non-portable Ada code involved a public domain math functions library. The functions used by the application worked correctly on the Rational. However on the 80386 system one of the functions produced erroneous results for certain input values. It was discovered that this math package had hard coded values for machine specific parameters. We did not try to determine if this was the cause of the problem as an alternative math functions library was available. This does point out the need for extensive test data, and a test mechanism, for "reusable" Ada packages.

There still seems to be a tremendous resistance to using Ada language features for embedded, real-time applications. Some of this comes from "experts" who have heard Ada is not

efficient enough, or just cannot support various real-time or "system" functions. This resistance is probably a positive sign. It used to be said that Ada was too inefficient for almost *all* applications, not just real-time applications. Unfortunately system specifics, such as a particular compiler, target, or any operating system involvement, tend to be forgotten or ignored. There are certainly systems which cannot come close to supporting time critical applications, but this does not mean all systems are that way. Much more surprising is the push by some Ada compiler vendors (and, less surprising, real-time kernel vendors) to promote special, non-Ada runtime systems. This could be seen as an attempt to distinguish their product, or provide a higher performance system where needed. However, it could also be viewed as an attempt to circumvent shortcomings in their runtime system implementation — which could lead to speculations of what else might be inefficient or poorly implemented in the system. The use of such systems greatly reduces the portability of the code and adds another complex system which has to be maintained.

Conclusions

We were able to implement a complex real time system in Ada, and did not have to resort to circumventing Ada language features or use a special, non-Ada run time system. This was a result of having hardware, and an Ada compiler and runtime system, with significantly more performance than was needed by the application. Futhermore, using the Ada tasking system allowed the initial debug and test of the code to be performed on the host development system, which was more accessable than the target system. This also allowed the debug and testing to begin before the target system was available. Another advantage of using Ada tasks and having sufficient performance margin was that it allowed the application to be implemented primarily by junior engineers. Some guidance was provided on implementing the interrupt handler and cyclic task execution. Otherwise they were able to use textbook tasking solutions, such as having tasks to coordinate exclusive access to resources. All of this indicates that as Ada compilers continue to mature the idea of leveraging of skills can be extended to the real-time arena.

THE VIEWGRAPH MATERIALS

FOR THE

R. LABAUGH PRESENTATION FOLLOW

EXPERIENCES WITH ADA IN AN EMBEDDED SYSTEM

Robert J. LaBaugh

MARTIN MARIETTA ASTRONAUTICS GROUP
SPACE SYSTEMS
DENVER, COLORADO

INTRODUCTION

EMBEDDED APPLICATION

- CONTROL OF EXPERIMENTAL ROBOT ARM

OBJECTIVES

- DEVELOP EMBEDDED APPLICATION IN ADA
  - EVALUATE LANGUAGE FEATURES
  - DETERMINE PERFORMANCE
- DEVELOP CONTROL SYSTEM BASED ON NASREM ARCHITECTURE IN ADA
- EXPERIMENT WITH CONTROL LAWS

BACKGROUND

ROBOT ARM WITH FIVE DEGREES OF FREEDOM

- SHOULDER, ELBOW, AND WRIST WITH ROLL, PITCH AND YAW

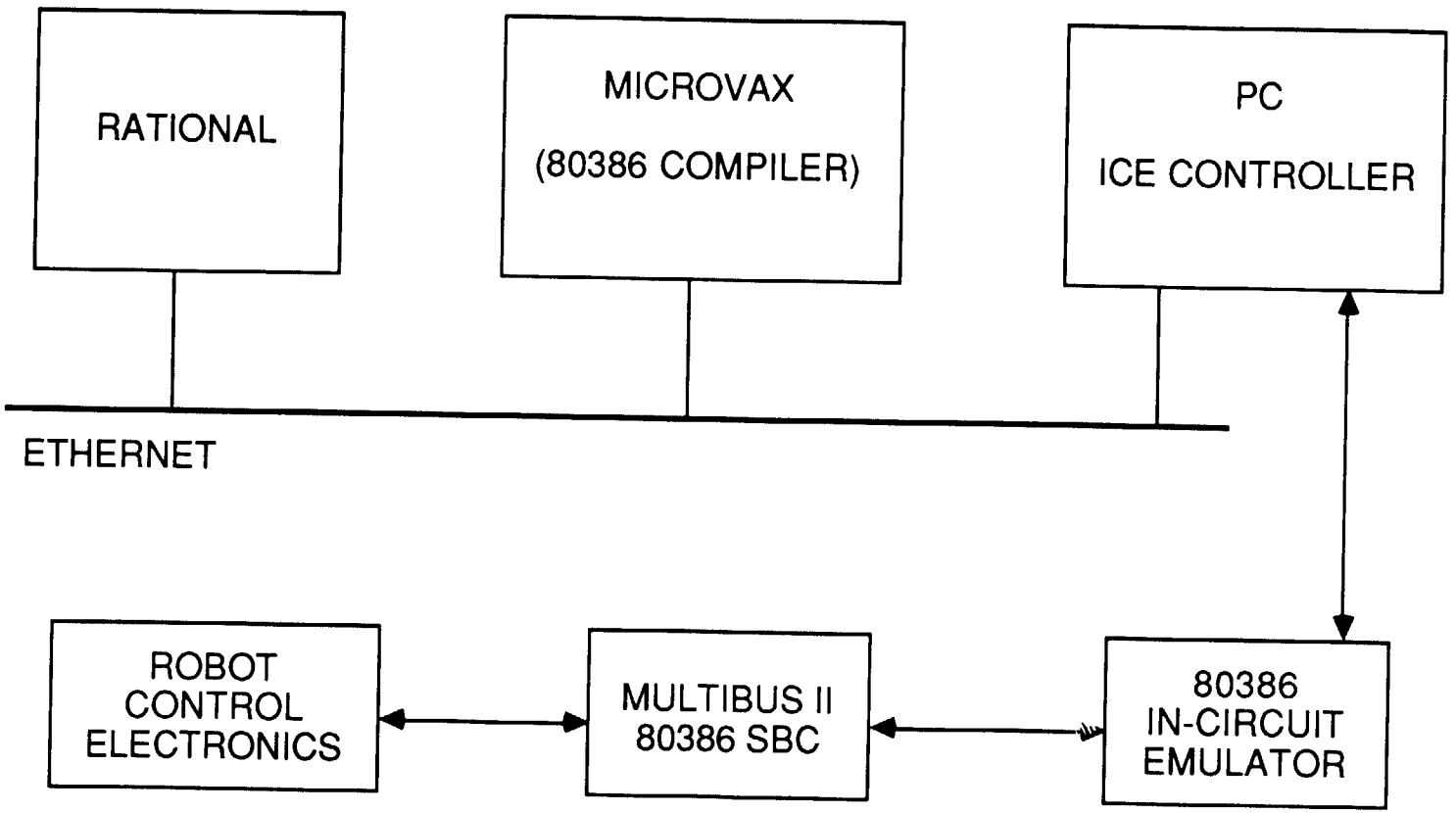
TARGET COMPUTER — INTEL 80386 SINGLE BOARD COMPUTER

- 80387 MATH COPROCESSOR
- 16 MHZ CLOCK FOR CPU AND COPROCESSOR
- 2 MB RAM
- RS-232 PORT

ADA COMPILER — DDC-I ADA 80386 PROTECTED MODE

- TARGETED TO BARE MACHINE
  - NO OPERATING SYSTEM
  - ADA RUNTIME SYSTEM PROVIDES COMPLETE CONTROL OF HARDWARE AND SOFTWARE
- TEXT_IO USES RS-232 PORT ON CPU BOARD
  - NON-BLOCKING I/O
  - USED FOR OPERATOR INTERFACE AND DEBUGGING

DEVELOPMENT SYSTEM



DEVELOPMENT APPROACH

INITIAL DEVELOPMENT AND TEST ON RATIONAL

- LOW LEVEL HARDWARE INTERFACE REPLACED WITH SIMULATION ROUTINES

SOURCE CODE DOWNLOADED TO MICROVAX

CROSS COMPILED AND LINKED ON MICROVAX

- ADA CROSS COMPILER FOR 80386 HOSTED ON VAX SYSTEM

LOAD MODULE MOVED TO IBM PC CLONE

- CONTROLER FOR IN-CIRCUIT EMULATOR
- USED TO LOAD AND CONTROL EXECUTION OF CODE IN TARGET SYSTEM

LESS THAN OPTIMAL ENVIRONMENT FOR DEBUG ON TARGET

- MOVING TOWARD COMPILING, DOWNLOADING, AND DEBUGGING FROM ONE TERMINAL
  - AVAILABLE NOW FOR SOME SYSTEMS, BUT LIMITED BY RS-232 SPEED

ADA FEATURES USED

ADA TASKS

- TASK RENDEZVOUS FOR SYNCHRONIZATION AND COMMUNICATION

INTERRUPT HANDLER CODED IN ADA

- ADDRESS CLAUSE FOR TASK ENTRY

PRIORITIZED TASKS

LOW_LEVEL_IO USED TO COMMUNICATE WITH HARDWARE INTERFACES

MACHINE CODE INSERTIONS

- USED TO DISABLE AND ENABLE INTERRUPTS
- NOT NEEDED IN INITIAL APPLICATION
  - USED FOR SAFE INITIALIZATION OF HARDWARE
- NEEDED AS MORE MULTIBUS II FEATURES USED
  - MULTIPLE HARDWARE ACCESSES PER LOGICAL OPERATION

APPLICATION

NASREM

- LAYERED, HIERARCHICAL CONTROL SYSTEM ARCHITECTURE
- COMMON INTERFACES AT ALL LEVELS
- DECREASING FREQUENCY OF OPERATION AS LEVELS INCREASE
- IMPLEMENTED WITH ADA TASKS

CONCENTRATED ON LOWEST TWO LEVELS

- SERVO LEVEL — SENDS MOTORS COMMANDS BASED ON ERROR BETWEEN CURRENT AND DESIRED POSITION
- PRIMITIVE LEVEL — DETERMINES EVENLY SPACED POINTS FOR SMOOTH MOTION, KINEMATIC TRANSFORMATIONS
- ELEMENTAL MOVE LEVEL — SIMPLE CANNED MOTION GENERATORS
- TASK LEVEL — SELECTS MOTION TO PERFORM

OPERATOR INTERFACE AND ROBOT CONTROL FUNCTIONS ON SAME (SINGLE) CPU

- 11 ADA TASKS

HARDWARE CLOCK INTERRUPT DRIVING CONTROL LOOP

RESULTS

ENTIRE APPLICATION CODED IN ADA

- NO SPECIAL KERNEL, SYSTEM CALLS, OR PRAGMAS
- "LOW LEVEL" CODE WRITTEN IN ADA
  - HARDWARE INTERFACE, DMA CONTROLLER, MULTIBUS II MESSAGE PASSING, ETC.

MORE THAN ADEQUATE PERFORMANCE FOR 50 HZ CONTROL LOOP

- ALL NASREM LEVELS ABLE TO EXECUTE IN 20 MSEC CYCLE

IN PROCESS OF SPLITTING CODE ONTO TWO CPUS

- ROBOT CONTROL
- OPERATOR INTERFACE
- INITIAL COMMUNICATION VIA SHARED MEMORY
  - ALSO EXPERIMENT WITH MULTIBUS II MESSAGE PASSING
- EXERCISE IN DISTRIBUTING APPLICATION TO HANDLE GROWTH

LESSONS LEARNED

ABILITY TO TEST AND DEBUG ON HOST DOES NOT ELIMINATE NEED FOR LOW LEVEL TESTING ON TARGET

- NEEDED TO BUILD CONFIDENCE IN BOTH THE HARDWARE AND SOFTWARE
  - CORRECTNESS OF INTERFACE DEFINITIONS
- NEEDED TO PROVIDE BASIS FOR TROUBLE-SHOOTING PROBLEMS
  - HARDWARE FAILURES, IMPROPER SYSTEM INITIALIZATION (POWER TURN ON)

PORTABILITY IS NOT AUTOMATIC WITH ADA

- DIFFERENCES IN TASKING IMPLEMENTATION IN RUNTIME SYSTEMS
  - TIME SLICING OF EQUAL PRIORITY TASKS ON RATIONAL
  - TIME SLICING NOT THE DEFAULT ON TARGET
- PUBLIC DOMAIN PACKAGES NEED SUPPORTING TEST DATA AND TEST MECHANISM
  - MATH FUNCTIONS LIBRARY — HARD CODED, MACHINE SPECIFIC PARAMETERS VS USE OF ADA LANGUAGE FEATURES — ERRONEOUS RESULTS FOR CERTAIN INPUT

LESSONS LEARNED (continued)

TREMENDOUS RESISTANCE TO USING ADA LANGUAGE FEATURES FOR EMBEDDED, REAL-TIME APPLICATIONS

(I.E. STANDARD ADA RUN TIME SYSTEM, INTERRUPT HANDLERS VIA TASK ENTRIES, LOW_LEVEL_IO)

- FROM "EXPERTS" WHO HAVE HEARD ADA IS NOT EFFICIENT ENOUGH OR CANNOT SUPPORT VARIOUS REAL-TIME/SYSTEMS FUNCTIONS
  - SPECIFICS (COMPILER, TARGET SYSTEM) TEND TO BE IGNORED
  - POSITIVE SIGN (REDUCED AREA OF "NON-USABLE")
- FROM COMPILER VENDORS AND REAL-TIME KERNEL VENDORS
  - PUSHING NON-STANDARD KERNELS/OPERATING SYSTEMS
  - ATTEMPT TO DISTINGUISH PRODUCT, OR CIRCUMVENT SHORTCOMINGS IN THEIR RUNTIME SYSTEM IMPLEMENTATION
  - COUNTERPRODUCTIVE — REDUCES PORTABILITY, ADDS ANOTHER COMPLEX SYSTEM WHICH HAS TO BE MAINTAINED

SUMMARY/CONCLUSIONS

COMPLEX REAL-TIME SYSTEM IMPLEMENTED ENTIRELY IN ADA

- HARDWARE AND COMPILER WITH SUFFICIENT PERFORMANCE

USING STANDARD ADA RUNTIME SYSTEM ALLOWED INITIAL DEBUG AND TEST ON THE HOST SYSTEM

- MORE ACCESSABLE THAN THE TARGET SYSTEM
- AVAILABLE EARLIER

PERFORMANCE MARGIN AND USE OF ADA FEATURES ALLOWED JUNIOR ENGINEERS TO IMPLEMENT THE APPLICATION

- GUIDANCE PROVIDED FOR IMPLEMENTING INTERRUPT HANDLER AND CYCLIC TASK OPERATION
- ABLE TO USE "TEXTBOOK" TASKING SOLUTIONS — EXCLUSIVE ACCESS TO RESOURCES

ADA COMPILERS ARE MATURING ENOUGH TO BE USED IN EMBEDDED SYSTEMS

PANEL #4

TOOLS

D. Drew, Unisys
P. Usavage, Jr, General Electric
J. F. Buser, Software Development Concepts

N91-10615

A Practical Approach to ObJect Based Requirements Analysis

Daniel W. Drew and Michael Bishop

Unisys, Houston Operations Division
600 Gemini Mail Code UO4C
Houston, Tx. 77058-2775
(713)-282-3664

Introduction

In the teaching of mathematics, problem statements are often used to provide exercises which require the students to apply the knowledge learned. The student must read a paragraph and determine first what the problem is, then apply the appropriate equation to find the answer. System development is analogous to solving math problem statements. There is the problem statement (requirements) which must be understood so that the right equation (design) can be applied for the solution.

If the study of mathematics emphasizes only the study of equations and how they are derived, the student will be ill-equipped to use that knowledge in practical applications. Similarly, design methods which do not have supporting methods for understanding requirements will prove difficult to use in practical system development.

The use of objects in design methodologies has provided a mechanism whereby software engineers can take fuller advantage of software engineering principles. However, these concept are just beginning to reach their full potential as we move them earlier into the lifecycle.

This paper presents an approach, developed at the Unisys Houston Operation Division, which supports the early identification of objects. This "domain oriented" analysis and development concept is based on entity relationship modeling and object data flow diagrams. These modeling techniques, based on the GOOD methodology developed at the Goddard Space Flight Center [4], support the translation of requirements into objects which represent the real-world problem domain. The goal is to establish a solid foundation of understanding before design begins, thereby, giving greater assurance that the system will do what is desired by the customer. The transition from requirements to object oriented design is also promoted by having requirements described in terms of objects.

Presented is a five step process by which objects are identified from the requirements to create a problem definition model. This process involves establishing a base line requirements list from which an object data flow diagram can be created. Entity-relationship modeling is used to facilitate the identification of objects from the requirements.

The paper concludes with an example of how semantic modeling may be used to improve the entity-relationship model and a brief discussion on how this approach might be used in a large scale development effort.

A Practical Approach to Object Based Requirements Analysis

1.0 Approach Overview

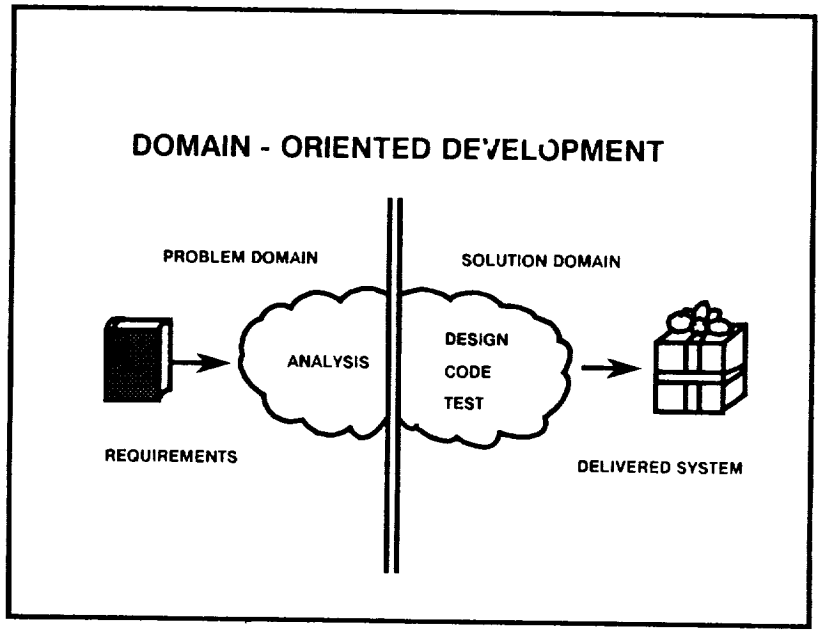
Following the principles of software engineering promotes a more pragmatic approach for system development. It requires a change in the overall concepts of how systems are created as well as new analysis and design methodologies.

1.1 Domain Oriented Development

For a design to be successful, there must be an understanding of the problem it is intended to solve. All too often problem definition is established in just enough detail to begin design and evolves as the design evolves. This can lead to unstructured systems which are hard to implement and expensive to maintain. To eliminate this problem software development can be divided into the problem and solution domains. The problem domain provides the foundation for all solution domain activities. A greater discipline is introduced into development giving greater assurance that the requirements (problem) are understood before a design (solution) is attempted.

Activities included in the problem domain are requirements generation and requirements analysis. The end product of requirements analysis is a problem definition model. This model becomes the foundation for all solution domain activities.

Activities included in the solution domain are preliminary design, detail design, code, and test. The end product is a delivered system which conforms to requirements.



1.2 The Mechanics of Requirements Analysis

Requirement analysis is concerned with establishing what a system is to do. This information must be documented in a form easily understood by all parties involved in development. The process for understanding a set of requirements requires an ordered set of steps which clarify original requirement statements and allow key information to be identified.

The remainder of this paper will show in detail an approach which is made up of the following steps:

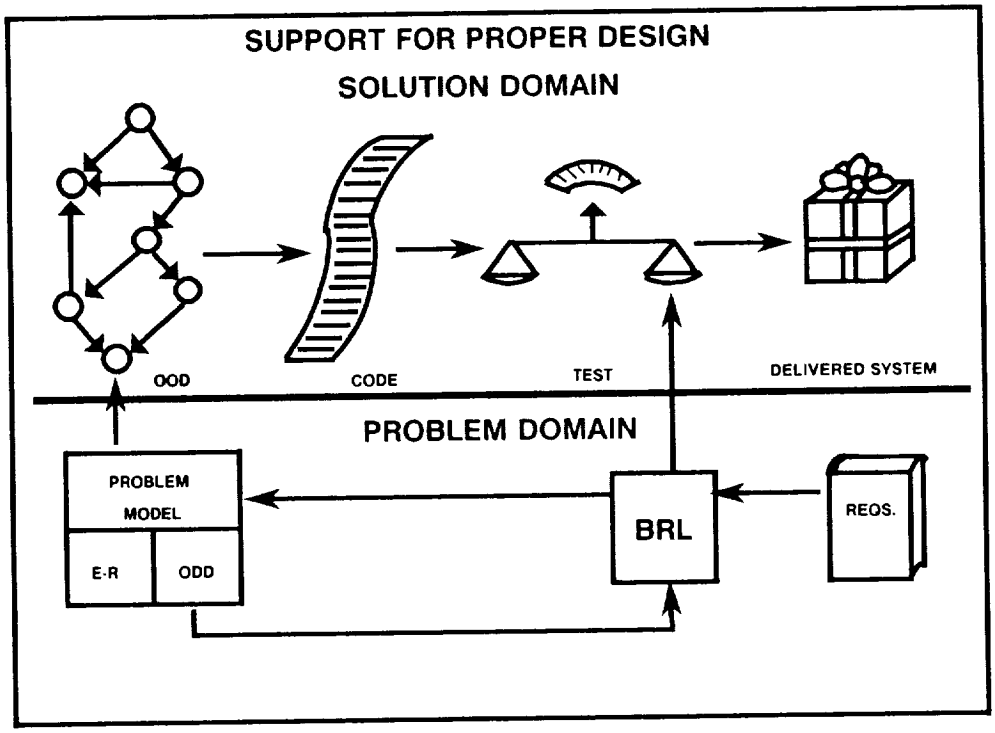
Step 1: Compile a notebook containing all requirement statements and information from other sources which might be pertinent to the problem.

Step 2: Rewrite the information from the notebook into concisely stated sentences. This establishes a baseline requirements list (BRL).

Step 3: Develop a static model of the problem from the BRL using entity relationship modeling. This model will facilitate the identification of objects.

Step 4: Identify the objects and develop a dynamic model of the problem from the entity relationship model using an object data flow diagram (ODD).

Step 5: Reorganize the BRL so that the statements are grouped by object.



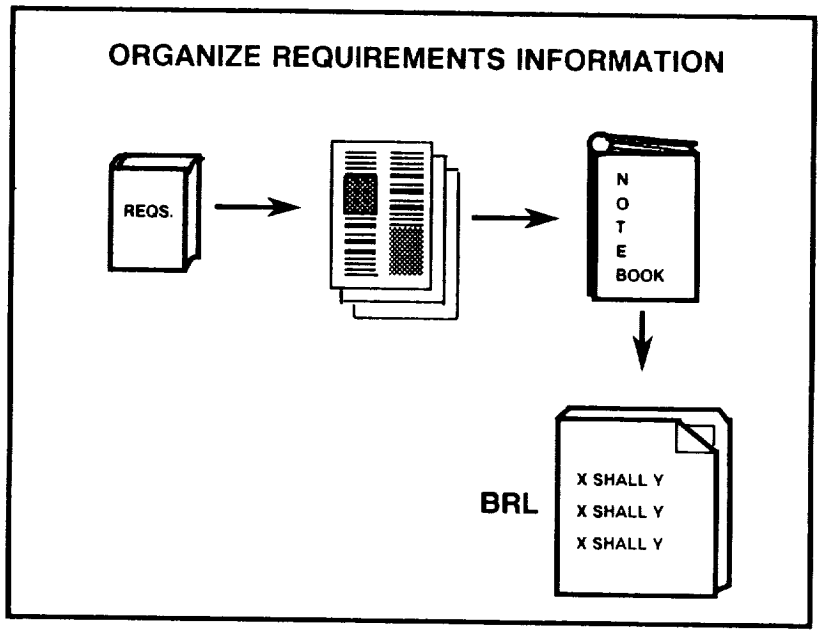
2.0 Step 1: Compiling an Information Notebook

Complete requirements information is essential in order to create the system the customer really wants. The purpose of step one is to gather all information which might have any possible bearing on what the system is to do. The actual process for this step will vary as to the sources of information available. For entirely new development projects, this is the initial step of requirements generation. Information must be compiled from many different sources. For enhancements to an existing system, this step is the identification and collection of requirements pertinent to the enhancements. Information sources would be the existing requirements document, design specifications, and interviews of current users and maintenance personnel. The end result is to have all the information available for design gathered in a single reference.

This approach was developed for a project which had an existing requirements document. The document was old and the system had undergone several major revisions. The notebook contained pages from the requirements document, information from a system closely related to the one being redesigned, and notes given by experts in the application. The end result was a collection of all available requirements information which then served as a single reference for analysis and design.

3.0 Step 2: Establishing a Baseline Requirements List

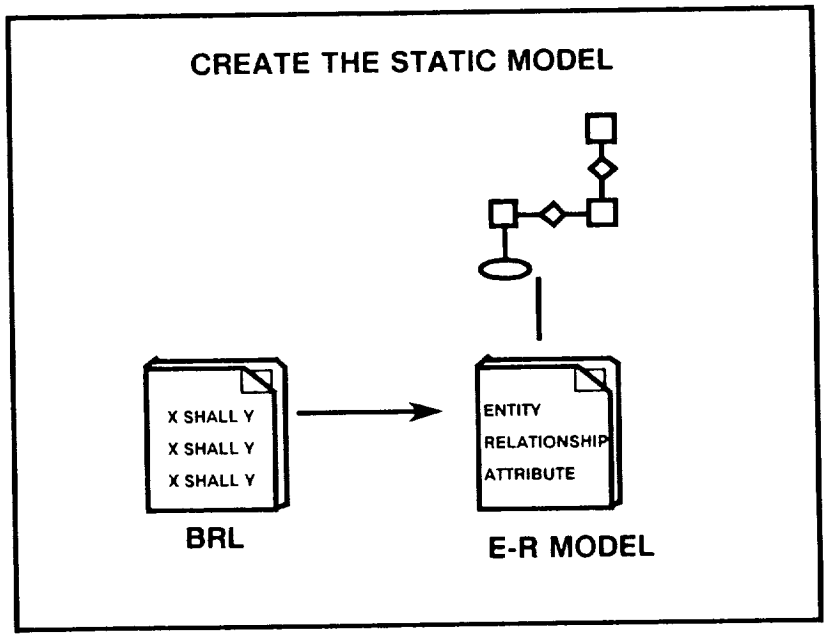
The resultant notebook contains all the information needed to create a model of what the system is to do. However, there is no meaningful structure. It is very difficult to determine: if the information is complete, if there is information that is not needed, or how the pieces of information relate to each other. A good organization of requirements is necessary in order to facilitate the extraction of entities, relationships, and attributes from the requirements and to develop the dynamic problem domain definition. The Baseline Requirements List (BRL) provides this needed structure. Each statement in the notebook is rewritten in a traditional "X shall Y" format where "X" is a noun or noun phrase and "Y" is some action the noun will perform. Rewriting in this form will force a greater understanding of each requirement piece. Ambiguous statements and statements which have no impact on what the system is to do can be easily recognized. Having all requirements stated as cause and effect also provide a solid platform for system testing.



4.0 Step 3: Developing a Static Problem Definition Model

A static model of the problem is the first component of the problem definition model. Its purpose is to give structure to the requirements information that will facilitate the identification of the dynamic properties of the system. A static model represents all the possible entities, with their attributes and relationships, described by the BRL. The development of a static model based on the requirements is an information representation problem. Therefore, it is reasonable to borrow modeling techniques from the DBMS world. Entity relationship modeling has been recommended by Mike Stark and Ed Seidewitz of the Goddard Space Flight Center [4] and Dr. Charles McKay of the University of Houston at Clear Lake [2] as an appropriate tool for the structuring of requirements information.

Issues of completeness in requirements can be addressed with this model. Incomplete requirements appear as dangling entities which have no relationships or as relationships without clearly defined entities. An entity without relationships may also indicate a requirement statement which does not belong to the problem. This type of inconsistency is identified and resolved in an iterative process of reviewing the requirement statements which make up the part of the entity-relationship model in question until all unusual model structures are resolved.



4.1 Entity-relationship Modeling

The approach promoted by this paper for entity-relationship modeling consists of the entity-relationship model creation phase, the entity dictionary, which provides entity definitions which will be used throughout the software lifecycle, and entity-relationship diagrams, which can be used to graphically depict portions of the entity dictionary. Object data-flow diagrams, which depict the dynamic problem definition are generated from the entity-relationship model and will be addressed in section 5.0. The remainder of this section presents in detail how an entity-relationship model is developed from the BRL.

A common example, a subset of a student registration system, will be presented with most of the topics in this section and in section 5 in order to help in understanding the concepts. The example will have the following requirements:

1. The system shall provide the capability to enter and maintain information regarding students.

2. The system shall provide the capability to enter and maintain information regarding the courses in which students are enrolled.

3. Student information shall include the student's name, age, major and social security number.

4. Course information shall include the course's name, department, room number, meeting time and days, name of the professor teaching the course, a list of students enrolled in the course, the number of students currently enrolled in the course and the maximum number of students allowed in the course.

5. A course shall be closed when the number of currently enrolled students reaches the maximum number of students allowed in the course. Otherwise, the course shall be considered open.

6. Students shall be allowed to enroll in an open course.

7. Students shall not be allowed to enroll in a closed course.

8. The system shall accept registration requests containing the name of a student and the name of the course in which he/she wishes to enroll.

9. Registration requests shall be processed in order to determine whether or not a student may enroll in the requested course.

4.2 Entity-Relationship Model Creation

The entity-relationship model creation phase consists of extracting entities, attributes and relationships from the requirements. During this phase, the requirements are assumed to be in the form of the BRL discussed in section 3.0.

4.2.1 Entity Extraction

Entities will appear as nouns in the requirement statements. Different types of noun phrases reveal different types of entities [3]. Common nouns, such as "terminal", "student" or "message", name a class of entities. Mass nouns and units of measure, such as "water", "matter" or "fuel", name a quality, activity, quantity or substance of the same. Proper nouns and nouns of direct reference, such as "my terminal", "George" or "syntax error advisory message", name specific instances of an entity class.

The requirements will not necessarily name all of the entities in the problem domain. Related entities may have to be found by looking through documentation, talking to people who have some expertise in the area, etc. For example, suppose that the problem domain consists of a bucket containing different types of fruit. The requirements may state that the job is to remove the apples and oranges from the bucket and place them in different piles. The entities in this problem domain, as shown by the requirements, are the apples, oranges and the bucket. However, there are other kinds of fruit that have to be considered when removing the apples and oranges (i.e. they must be discarded). Those other fruits are part of the problem domain and therefore are entities in the problem domain.

There is another case in which entities are not explicitly named in the requirements. Suppose that the requirements in the apples and oranges problem also state that someone is to be notified when a spoiled apple is found in the bucket. This new requirement introduces two new entities, a spoiled apple and a notification that a spoiled apple has been found. There is a gap in the problem domain model between the spoiled apple and the notification of the spoiled apple. This gap is filled by an entity that represents the event that is characterized by finding the spoiled apple. The event entity is related to the notification entity in that someone is to be notified in the event that a spoiled apple is found.

Entities are either internal or external. Internal entities have an existence only within the scope of the problem domain. External entities have an existence outside the scope of the problem domain. The concept of internal and external entities is easier to consider if the problem domain is thought of as a "black box." Internal entities cannot be seen outside of the box but external entities can be seen entering or leaving the box.

In the student registration example, the requirements yield the following entities:

From requirement 1: Student

From requirement 2: Course, Student

From requirement 3: Student

From requirement 4: Department, Professor, Course_Roster, Course

From requirement 5: Course, Closed_Course, Open_Course, Student

From requirement 6: Student, Open_Course

From requirement 7: Student, Closed_Course

From requirement 8: Registration_Request

From requirement 9: Course, Registration_Request, Student

The Course_Roster in requirement 4 is the list of students enrolled in a course.

4.2.2 Attribute Extraction

Attributes usually appear in the requirements as information concerning entities. The following attributes are named in the requirements:

Student:	Student_Name, Age, Major, SS_Number
Course:	Course_Name, Current_Size, Max_Size, Time, Days,Room_Number,Professor_Name, Department_Name
Professor:	Professor_Name
Department:	Department_Name
Registration_Request:	Student_Name, Course_Name

4.2.3 Relationship Extraction

Relationships appear in the requirements as associations between pairs of entities, entities and attributes or relationships and attributes. The student registration requirements show the following relationships:

Requirement 2:	Is_Enrolled_In (1:m) between Student and Course.

Requirement 4:
Includes (1:1)/Is_A_Part_Of (1:1)
between Course and Course_Roster;
Includes (1:m)/Is_A_Part_Of (m:1)
between Department and Course, Professor;
Is_A_List_Of (1:m)/Is_A_Member_Of (m:1)
between Course_Roster and Student;
Teaches (1:1)/Is_Taught_By (1:1)
between Professor and Course.

Requirement 5:
Is_A_Type_Of (1:1)
between Closed_Course and Course,
between Open_Course and Course;
Is_An_Instance_Of (1:1)
between Course and Open_Course or Closed_Course.

Requirement 6:
May_Enroll_In (1:m)
between Student and Open_Course.

Requirement 7:
May_Not_Enroll_In (1:m)
between Student and Closed_Course.

Requirement 9:
References (1:1) / Is_Referenced_By (1:m)
between Registration_Request and Student,
between Registration_Request and Course.

A slash between two relationship names indicates a pair of symmetric, oppositely-directed relationships. In requirement 4, Course includes Course_Roster and conversely, Course_Roster is a part of Course. The mapping class of the relationship is indicated in parentheses.

4.3 Entity Dictionary

The entity dictionary provides a means of describing the entities that are part of the problem domain. A data structure that is useful for representing the entity dictionary is the frame [4], a form of knowledge representation developed by Marvin Minsky. A frame is a generalized property list containing a list of symbols with their associated property names and values [5].

The following is an example of entity entries in the student registration entity dictionary.

Closed_Course (Entity)
    Rqmt_Numbers 5, 7
    Scope External
    Is_A_Type_Of Course

Course (Entity)
    Attributes Course_Name, Department_Name, Room_Number, Time, Days,
                Professor_Name, Current_Size, Max_Size
    Rqmt_Numbers 2, 4, 5, 9
    Scope External
    Is_Taught_By Professor
    Is_A_Part_Of Department
    Is_An_Instance_Of Open_Course, Closed_Course

```
        Is_Referenced_By Registration_Request
        Includes Course_Roster

Course_Roster (Entity)
        Rqmt_Numbers 4
        Scope Internal
        Is_A_Part_Of Course
        Is_A_List_Of Student

Department (Entity)
        Attributes Department_Name
        Rqmt_Numbers 4
        Scope External
        Includes Course, Professor

Open_Course (Entity)
        Rqmt_Numbers 5, 6
        Scope External
        Is_A_Type_Of Course

Professor (Entity)
        Attributes Professor_Name
        Rqmt_Numbers 4
        Scope External
        Teaches Course
        Is_A_Part_Of Department

Registration_Request (Entity)
        Attributes Student_Name, Course_Name
        Rqmt_Numbers 8, 9
        Scope Internal
        References Student, Course;

Student (Entity)
        Attributes Student_Name, Age, Major, SS_Number
        Rqmt_Numbers 1, 2, 3, 5, 6, 7, 9
        Scope External
        Is_Enrolled_In Course
        Is_A_Member_Of Course_Roster
        May_Enroll_In Open_Course
        May_Not_Enroll_In Closed_Course
        Is_Referenced_By Registration_Request
```

The entity dictionary can be extended to include attributes. The following is an example of some of the attribute entries in the student registration entity dictionary.

```
Course_Name (Attribute)
        Is_An_Attribute_Of Course
        Rqmt_Numbers 4, 8
        Domain String

Days (Attribute)
        Is_An_Attribute_Of Course
        Rqmt_Numbers 4
```

Domain Character
Values M, T, W, R, F, MWF, TR, MW

Student_Age (Attribute)
Is_An_Attribute_Of Student
Rqmt_Numbers 3
Domain Integer
Range 16..100

Time (Attribute)
Is_An_Attribute_Of Course
Rqmt_Numbers 4
Domain Character
Length 5
Range 08:00..19:00

4.4 Entity-Relationship Diagrams

Entity-relationship diagrams are used to graphically depict a part of the problem domain. Attempts were made to split the problem domain into parts by using a levelling technique in which the upper levels in the problem domain consist of "aggregate entities" with the actual problem domain entities at the lower levels. Unfortunately, there was not much progress in this endeavor and therefore a single-level description of the problem domain was created. Since a diagram showing the entire problem domain would be cumbersome, it is better to use the entity dictionary as the problem domain definition with entity-relationship diagrams being generated to map parts needing greater clarification. [4].

In the entity-relationship diagram, entities are represented by rectangles and relationships by diamond-shaped boxes [1]. Attributes are listed next to the rectangle representing the entity. The arrows indicate the direction of relationships. A double-h aded arrow indicates the 1:m, m:1 or m:n mapping class.

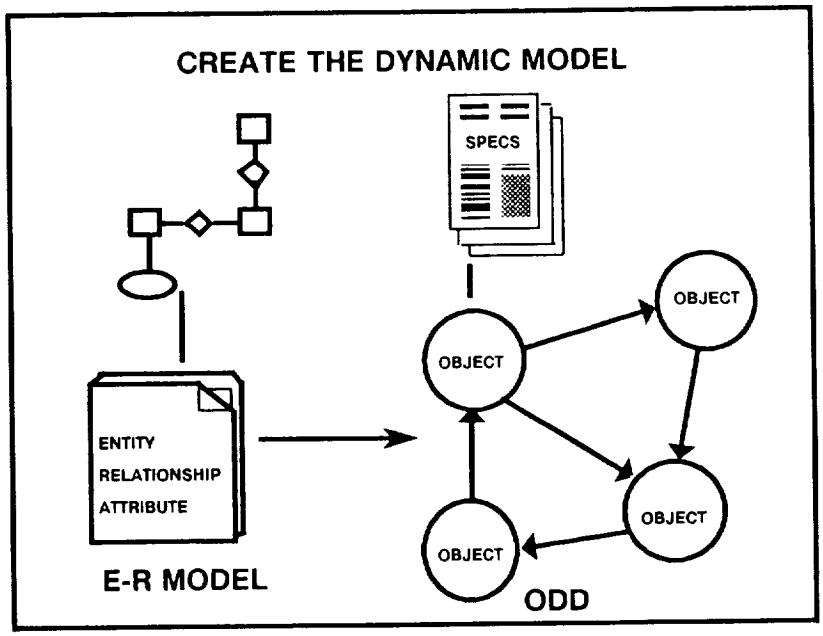
Entity-relationship diagrams can be generated in order to graphically map the problem domain onto one or more requirements or to show the problem domain from the perspective of a particular entity. In the latter application, it is useful to state the "order" of the diagram. A first-order entity-relationship diagram shows the central entity (the entity from whose perspective the problem domain is being viewed) and its relationships to surrounding entities. A second-order diagram shows the central entity, its relationships to surrounding entities and the relationships of each of the surrounding entities to its surrounding entities.

Sample entity-relationship diagrams for the student registration system are shown in appendix A.

5.0 Step 4: Developing a Dynamic Problem Definition Model

The second and concluding component of the problem definition model is the dynamic model of the problem. It is through this model that data flow and control, as described by the requirements, is represented. An object data flow diagram (ODD) is used to model the dynamic properties of the problem [4]. An ODD is very similar to a data flow diagram from Yourdon structured analysis techniques. The chief difference lies in what the bubbles represent. For an ODD the bubbles are objects. Since data is encapsulated in objects, there will not be any data stores.

The remainder of this section will present in detail how an ODD is derived from an entity-relationship model.



5.1 Identifying Problem Domain Objects

An object is a unique instance of an abstract data type which is a set of data and operations associated with that data. In order to identify the objects in the problem domain, first find all of the major abstract data types apparent in the problem domain and use an object to manage each one. The abstract data types are represented by entities that do not have an Is_A_Type_Of, Is_An_Instance_Of, Is_A_List_Of or Is_A_Set_Of relationship to another entity. These entities are at the highest level of abstraction for entities of a particular type. In the student registration problem, those entities are Course, Department, Professor, Registration_Request and Student. Each one of these entity classes will have an object to manage it. These candidate objects are Course_Folder, Department_Folder, Professor_Folder, Registration and Student_Folder.

The next step is to find all entity classes associated through the Is_A_Type_Of, Is_A_List_Of or Is_A_Set_Of relationship with the entity classes found in the first step. In the example, Open_Course and Closed_Course are associated with Course through the Is_A_Type_Of relationship and Course_Roster is associated with Student via the Is_A_List_Of relationship. The following objects and associated entities can be identified thus far:

Course_Folder:	Course, Open_Course, Closed_Course
Department_Folder:	Department
Professor_Folder:	Professor
Registration:	Registration_Request
Student_Folder:	Student, Course_Roster

The entity classes listed for each object represent the abstract data types provided by that object. At this point, it is possible to determine the set of requirements satisfied by each object. This is done by consulting the entity dictionary and finding the requirement statement numbers for each of the entities associated with each object. Applying this process to the five objects in the student registration system shows the requirements satisfied by each object:

Course_Folder:	Requirements 2, 4, 5, 6, 7 and 9
Department_Folder:	Requirement 4
Professor_Folder:	Requirement 4
Registration:	Requirements 8 and 9
Student_Folder:	Requirements 1, 2, 3, 4, 5, 6, 7 and 9

The requirement sets for each object are not disjoint. The shared requirements (numbers 2, 4, 5, 6, 7 and 9 in our example) describe the relationships between entities of different types. These relationships, in turn, describe the interfaces between different objects.

In order to complete the definition of the problem domain objects, find all relationships between entities of different types and add each member of the corresponding entity pairs to the appropriate object. For example, because of the relationship "Student Is_Enrolled_In Course", there is an interaction between the Student_Folder and Course_Folder objects. To show this interaction, add the entity Student to the Course_Folder object and Course to Student_Folder. One exception to this procedure occurs when one member of the entity pair is a generalized entity class. An example of this is the relationship "Student May_Enroll_In Open_Course". Since "Open_Course Is_A_Type_Of Course", add Course to Student_Folder instead of Open_Course. This procedure results in the following set of objects and associated entities:

Course_Folder:	Course, Open_Course, Closed_Course, Course_Roster, Student, Department, Professor, Registration_Request
Department_Folder:	Department, Course, Professor
Professor_Folder:	Professor, Department, Course
Registration:	Registration_Request, Student, CourseStudent, Course_Roster, Course, Registration_Request

Having identified the problem domain objects and their associated entities, an object data-flow diagram can be generated.

5.2 Generating Object Data-Flow Diagrams

Generating an object data-flow diagram based on a set of problem domain objects is simply a matter of finding entities common to pairs of objects. For example, the entities that Course_Folder and Student_Folder have in common are Student, Course_Roster and Course. Those common entities represent interfaces between Course_Folder and Student_Folder. On the object data-flow diagram, the interfaces are represented by drawing a line between the two rectangles representing the objects and labeling the line with the names of the common entities. The object data-flow diagram representing the objects from section 5.1 is in Appendix B.

The problem domain objects identified can be formally documented (in terms of the entities used and produced) by adding them to the entity dictionary:

Course_Folder (Object)
Rqmt_Numbers 2, 4, 5, 6, 7, 9
Uses Department, Course_Roster, Student, Registration_Request, Professor
Produces Course

Department_Folder (Object)
Rqmt_Numbers 4
Uses Professor, Course
Produces Department

Professor_Folder (Object)
Rqmt_Numbers 4
Uses Department, Course
Produces Professor

Registration (Object)
Rqmt_Numbers 8, 9
Uses Student, Course
Produces Registration_Request

Student_Folder (Object)
Rqmt_Numbers 1, 2, 3, 4, 5, 6, 7, 9
Uses Course, Registration_Request
Produces Student, Course_Roster

5.3 Object Names

The names given to objects play a key role in the development and understanding of the ODD. Naming objects is possibly the most difficult task in requirements analysis. The objects supply the framework for the representation of information and the eventual design. Therefore, their names must convey a concise meaning of the abstraction.

Object names are always nouns or noun phrases. This facilitates using the objects as a structure which can be used to explain action. It should be obvious from the name what real world object is represented. It is very difficult if not impossible to pick object names which do not bias design toward a particular direction. Therefore, this fact must be understood and preconceived notions must be addressed when a name is chosen. The name must be broad enough to encompass all the details associated with an object. Operations found within an object should not contradict the implied meaning of the object's name.

6.0 Step 5: Reorganization of the BRL

The entity-relationship model and ODD provide a complete problem definition model. Furthermore, the ODD serves as a platform to launch into an object oriented design. The last step for the problem domain segment of development is to go back and group the statements in the BRL under headings which represent the objects they support.

The objects are the main organizational structure for the system. Re-grouping the requirements will help the designers to find the additional detail needed to continue development. It will help the testers create test procedures aligned along object boundaries. It will simplify the traceability of requirements to design for the designer, tester, and maintainer. In short, having the requirements document reflect the structure of the emerging design will provide a high level of continuity throughout the system's lifecycle.

7.0 Enhancements to Problem Definition Modeling

Requirements analysis is a specific application of an information representation problem. As current modeling techniques evolve, it is reasonable to expect improvements in the approach taken in problem definition modeling. Semantic data models are currently being introduced for use in modeling data bases. They provide a richer medium for the representation of information. This section describes how semantic modeling can be used to enhance the entity relationship model.

Semantic data models allow designers to represent the entities of interest in an application in a way that more closely resembles the view the user has of these entities [6]. Semantic data models provide abstraction constructs that can be used to capture some of the meaning of the user application.

The semantic entity-relationship model introduced in this section features the abstraction constructs provided by the semantic and hyper-semantic [6] data models and allows the analyst to further define the problem by stating the meaning of relationships between entities in the problem domain.

7.1 Modeling Primitives

Modeling primitives are atomic relationships whose meanings cannot be defined as a composition of other meanings. They form the basis on which other relationships can be defined. Modeling primitives can be grouped into relationship classes which correspond to the abstraction constructs of the hyper-semantic data model. The modeling constructs of the hyper-semantic data model and their associated relationship names include [6]

Generalization: Similar entities are abstracted into a higher level entity-class. Relationship: Is_A_Type_Of.

Classification: Specific instances are considered as a higher level entity-class. Relationship: Is_An_Instance_Of.

Aggregation: An entity is related to the components that make it up. Relationship: Includes / Is_A_Part_Of.

Set Membership: Several entities are considered as a higher level set entity-class.Relationships: Is_A_Set_Of / Is_A_Member_Of.

List Membership: Several entities are considered as a higher level list entity-class. Relationships: Is_A_List_Of / Is_A_Member_Of.

Constraint: A restriction is placed on some aspect of an entity or relationship. Relationship: Is_A_Constraint_On.

Heuristic: An information derivation mechanism is attached. Relationship: Is_A_Heuristic_On.

Synchronous Temporal: Specific entities are related by synchronous characteristics and considered as a higher-level entity-class. Relationships: Is_A_Predecessor_Of / Is_A_Successor_Of.

Asynchronous Temporal: Specific entities are related by asynchronous characteristics and considered as a higher-level entity-class. Relationships: Initiates / Is_Initiated_By.

Equivalence: Specific instances of an entity-class are asserted to be equivalent. Relationships: Is_Equivalent_To.

The slash within the relationship names indicates two oppositely-directed relationships.

7.2 Semantic Relationship Definition

The semantic entity-relationship model provides a construct that allows the analyst to define the meaning of a relationship. This construct can be used to define a relationship in terms of other relationships and modeling primitives and to define the restriction class of a relationship.

A relationship between entity classes A and B is restricted if instances of type A may only be related to certain instances of type B based on a condition. The relationship is existence restricted if instances of type A may only to be related to those instances of type B for which they depend on their existence [7].

In order to walk through a short example of a relationship definition, consider the Is_Enrolled_In relationship between Student and Course. The objective is to state what is meant by the phrase, "a student is enrolled in a course." The course roster may be used in order to determine if a particular student is enrolled in a particular course. Remember from section 4.2 that a course roster is a list of students enrolled in a course. Therefore, a student is enrolled in a course if the student is on the course roster. The relationship is written in the following form using the semantic relationship definition construct:

```
entity class Course, Student, Course_Roster;

relationship Is_Enrolled_In (entity_instance, entity_instance);

Student Is_Enrolled_In Course if
        CR Is_An_Instance_Of Course_Roster and
        Course Includes CR and
        Student Is_A_Member_Of CR;
```

The relationship statement declares Is_Enrolled_In as a relationship between two entity instances. Therefore, the definition of the Is_Enrolled_In relationship between Student and Course is concerned with an instance of Student and an instance of Course.

The first clause within the relationship definition, "CR Is_An_Instance_Of Course_Roster", defines an entity CR which is an instance of entity class Course_Roster. The second clause, "Course Includes CR", associates CR with the particular instance of Course with which the relationship is invoked. The third clause states that the instance of Student with which the relationship is invoked must be a member of the course roster CR in order for the Is_Enrolled_In relationship to be satisfied.

The relationship is invoked by replacing Student and Course with appropriate instances, for example "George Is_Enrolled_In Physics". In this invocation of the relationship, CR is the course roster for Physics and the relationship is satisfied if George is on that roster.

The semantic relationship definition construct can be thought of as "infix Prolog". In fact, it is rather easy to convert the above example into Prolog:

```
is_enrolled_in (Student, Course):-
        is_an_instance_of (CR, Course_Roster),
        includes (Course, CR),
        is_a_member_of (Student, CR).
```

If one could "code" the modeling primitives in Prolog and generate the appropriate Prolog declarations, it would be possible to execute a problem domain model. This may be useful in ensuring that the problem domain model is correct before going on to create objects and initiate design. This process is analogous to executing a design before implementation.

8.0 Considerations For Large Projects

This paper is based on a small project projected to be only 10,000 lines of code. An important question to ask is, "How will this approach support the development of a large system of 500,000 lines or greater?"

The basic approach is good for any size project. What complicates larger systems is the large number of requirements to be considered. It may not be practical or even possible to examine all the requirements at the same time as was done for this project.

To resolve this problem, approach the requirements as layers of abstraction. Read through the document and extract those statements which define a very high level view of the system. Apply the approach presented in this paper to produce a problem definition model for this high level abstraction. Now begin an iterative process of stripping off layers of detail for each object identified in the previous level of abstraction and create a problem definition model. Use the approach presented in this paper for each iteration.

As each layer of abstraction is added to the model, check the preceding layer to assure that the objects and interfaces already established still hold true. If there are inconsistencies, make the necessary adjustments and continue with the process.

A Practical Approach to Object Based Requirements Analysis

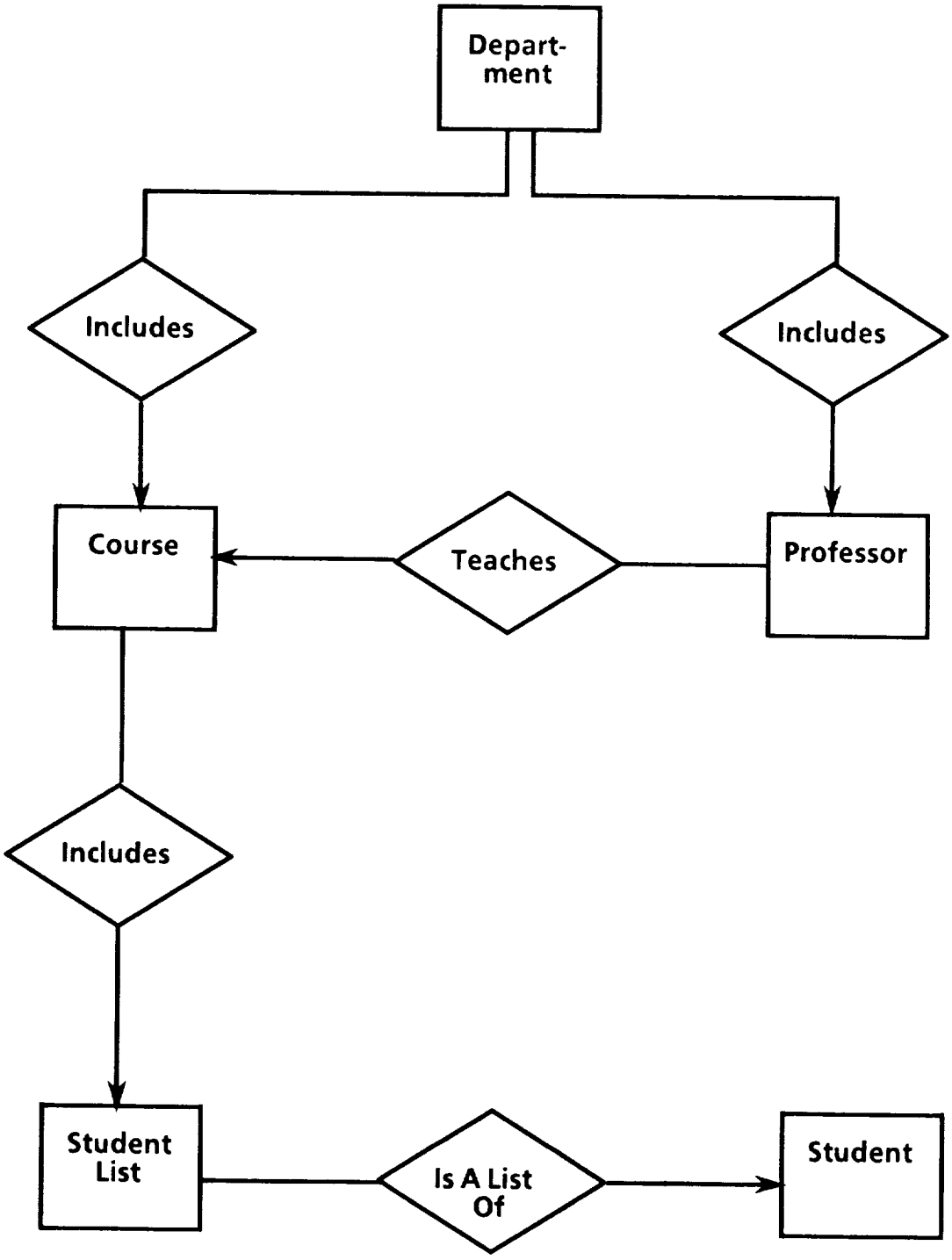
Summary

Students who spend all their time understanding math equations without applying them to problem statements will be ill-equipped to solve real world problems. System developers who possess the latest techniques in system design but have inadequate approaches to requirements analysis are destined to create wonderful designs which solve the wrong problem. The approach in this paper is a beginning to the application of modern analysis techniques rooted in the theoretical foundation of software engineering. A pragmatic approach allows for better conformance to those requirements in design. A model based on objects permits closer adherence to software engineering principles earlier in the lifecycle. It is not always easy to see objects in the requirements. Use of the entity-relationship model eases this problem by structuring the information in a form more conducive to object recognition.

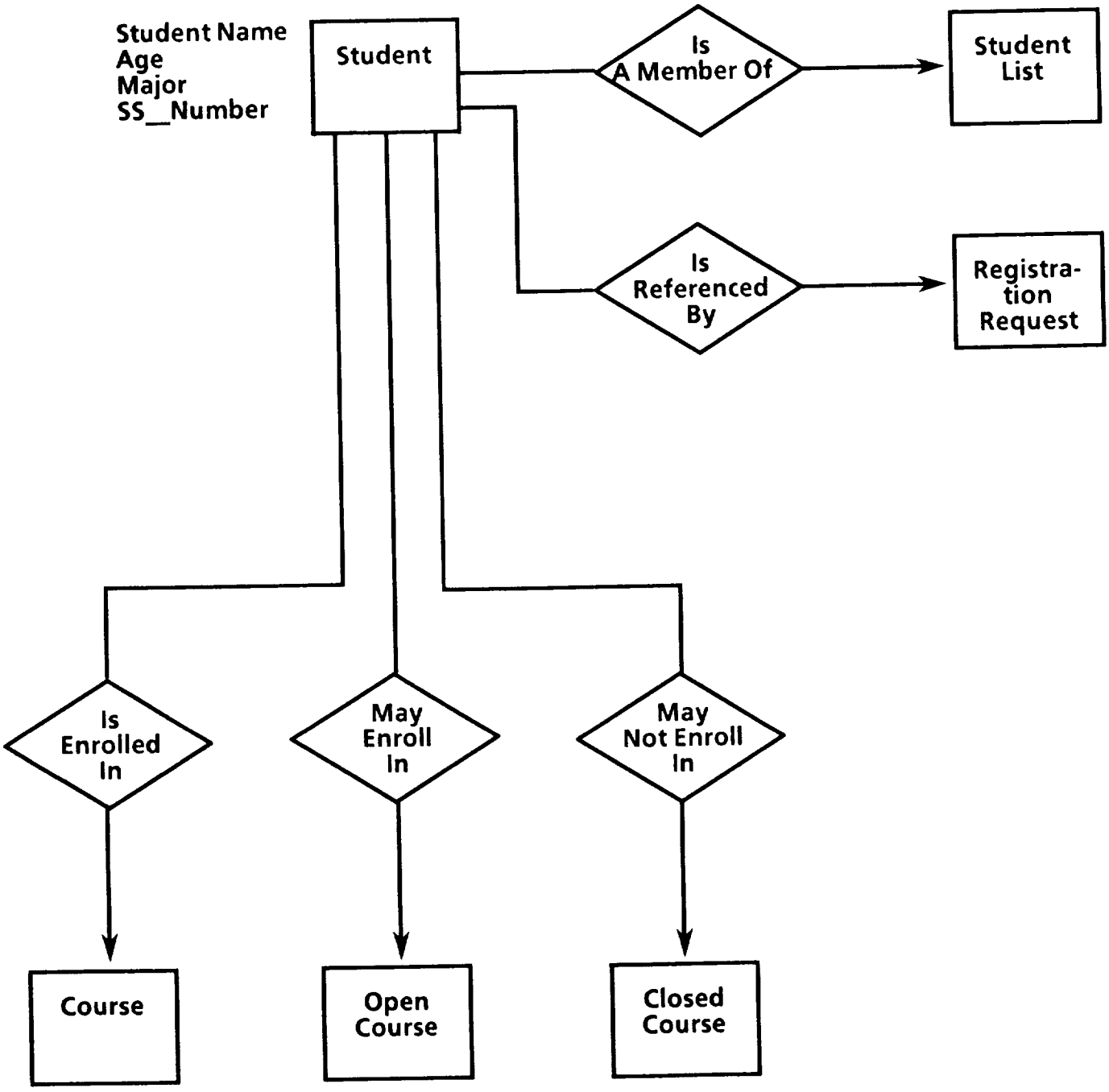
D. Drew
Unisys

Appendix A. Sample Entity_Relationship Diagrams

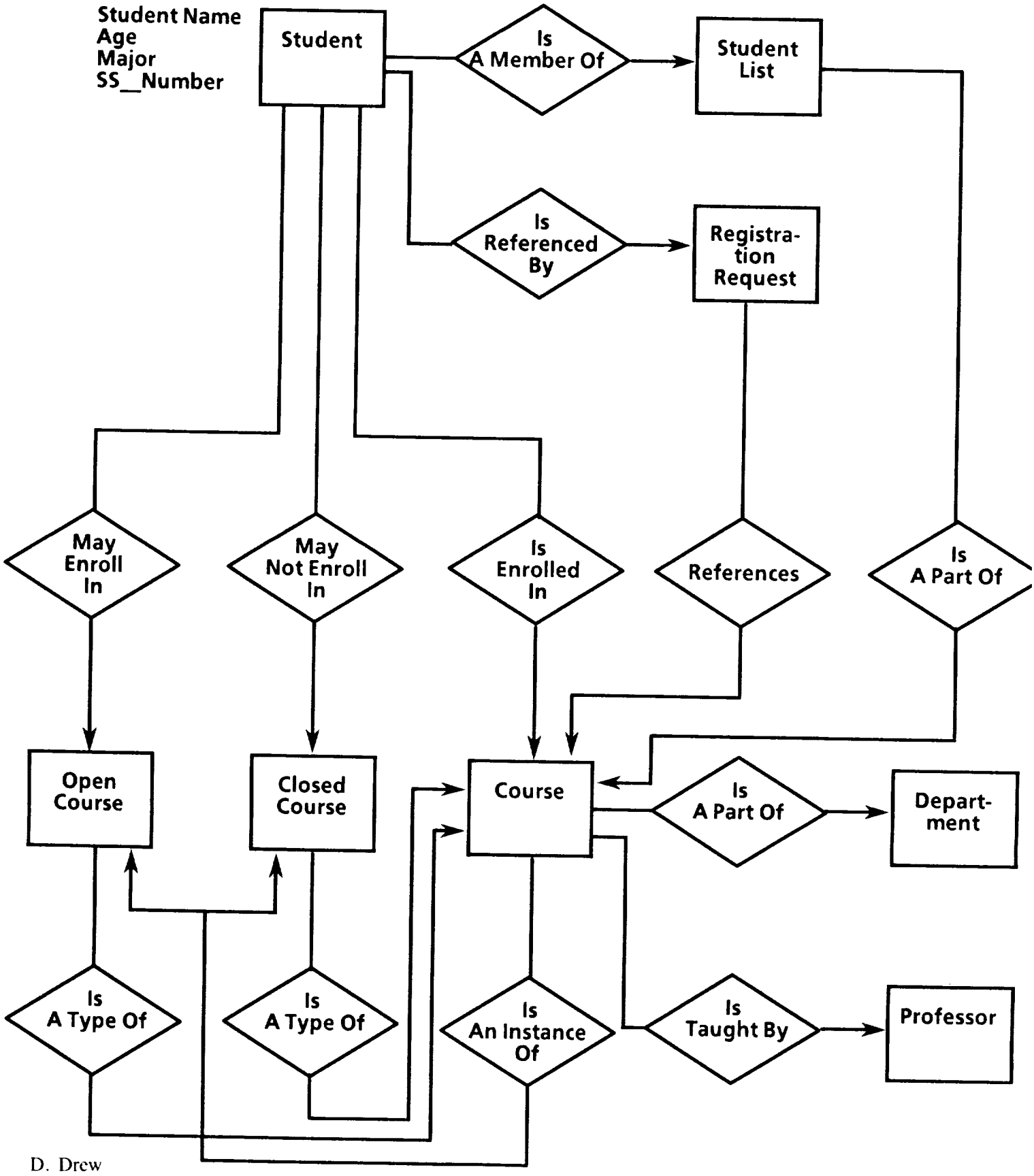
E-R DIAGRAM FOR STUDENT REGISTRATION REQUIREMENT 4



First order diagram for entity Student

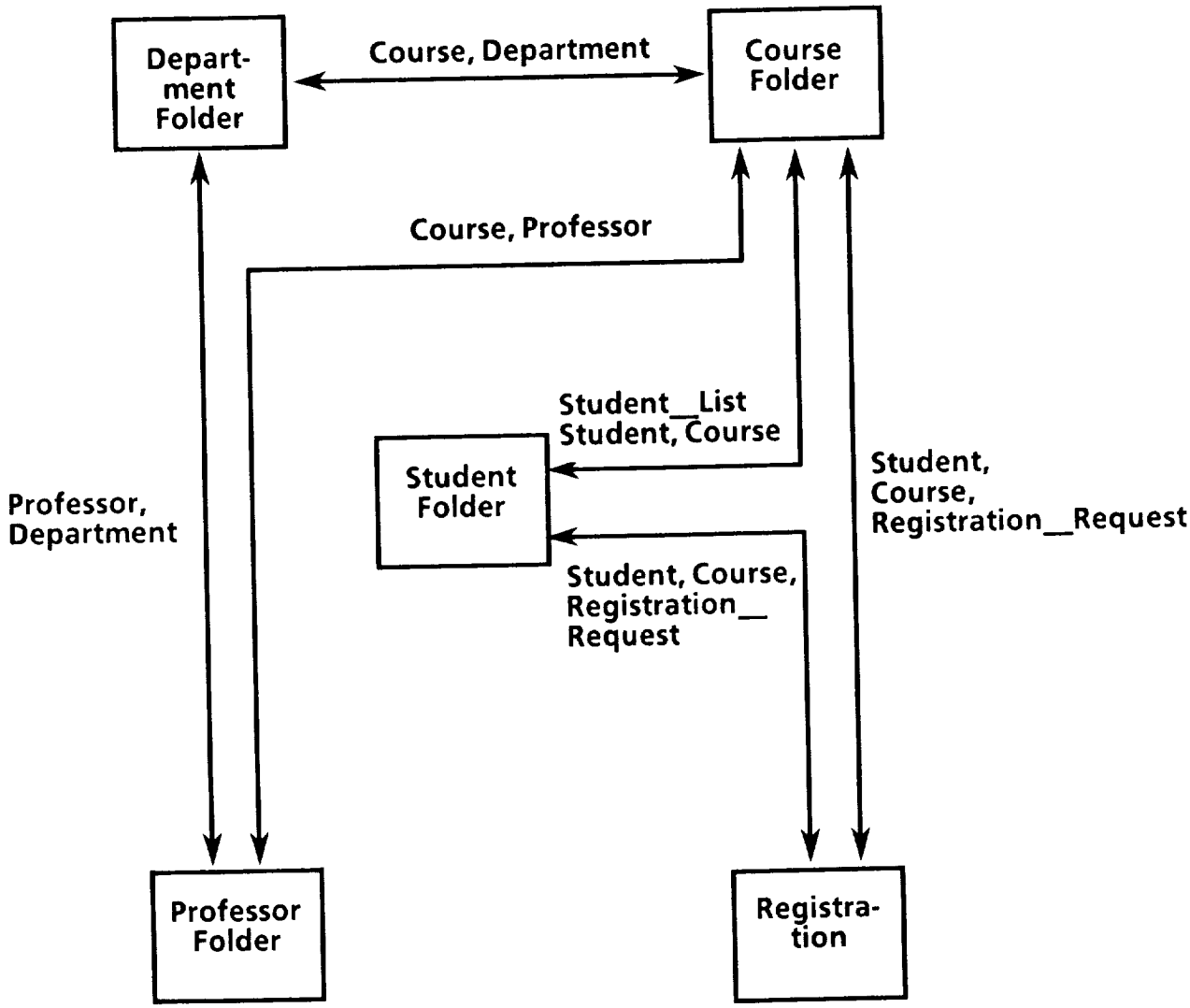


Second order diagram for entity Student



Appendix B. Sample Object Data-Flow Diagram

Object data-flow diagram for student registration system



References

[1] Chen, Peter P., "The Entity-Relationship Model - Toward a Unified View of Data", ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, pp. 9-36.

[2] McKay, Charles W., "A Perspective and Overview of Software Engineering", a seminar sponsored by the Software Engineering Research Center at the University of Houston at Clear Lake.

[3] Booch, Grady, Software Engineering With Ada, Second Edition, The Benjamin/Cummings Publishing Company, Inc., 1987.

[4] Stark, Mike and Seidewitz, Ed, "Towards A General Object-Oriented Ada Lifecycle", Goddard Space Flight Center, Greenbelt, Md., March 1987.

[5] Winston, Patrick H. and Horn, Berthold K. P., Lisp, Addison-Wesley Publishing Company, 1981.

[6] Potter, Walter D. and Trueblood, Robert P., "Traditional, Semantic, and Hyper-Semantic Approaches to Data Modeling", Computer, June 1988, pp. 53-63.

[7] Webre, Neil W., "An Extended Entity-Relationship Model And Its Use On A Defense Project", Entity-Relationship Approach To Information Modeling And Analysis, ed. by Peter P. Chen, Elsevier Science Publishing Company, 1983.

Biographical Sketch

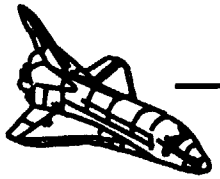
Daniel Drew has worked 13 years in the computer industry. After graduating with a B.S. degree in Computer Science from Texas A&M University, he spent 10 years developing Supervisory, Control, and Data Acquisition (SCADA) systems for oil pipeline control and automated oil field production. He has spent the last three years in Aerospace as a system designer and currently as section supervisor at the Unisys, Houston Operations Division. The section he supervises is working the first Ada pilot project attempted at the Unisys Houston site. Mr. Drew is a member of the IEEE Computer Society, Clear Lake Chapter of SigAda, and National SigAda.

Michael Bishop has worked as a programmer and systems analyst in the aerospace industry for over four years. Mr Bishop is currently employed at the Unisys Houston Operation Division where he has been developing an entity-relationship methodology suitable for a wide range of applications as part of an Ada pilot project. Previously, Mr. Bishop worked at Ford Aerospace and Unisys on the MAST project, a database application concerned with the management of Space Shuttle downlink and uplink data. Mr. Bishop received his bachelor's degree in Computer Science in 1984 from the University of Houston and is currently pursuing a master's degree at the University of Houston's Clear Lake campus.

THE VIEWGRAPH MATERIALS

FOR THE

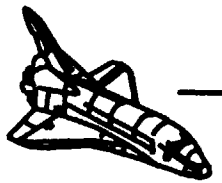
D. DREW PRESENTATION FOLLOW



UNISYS
Houston Operations

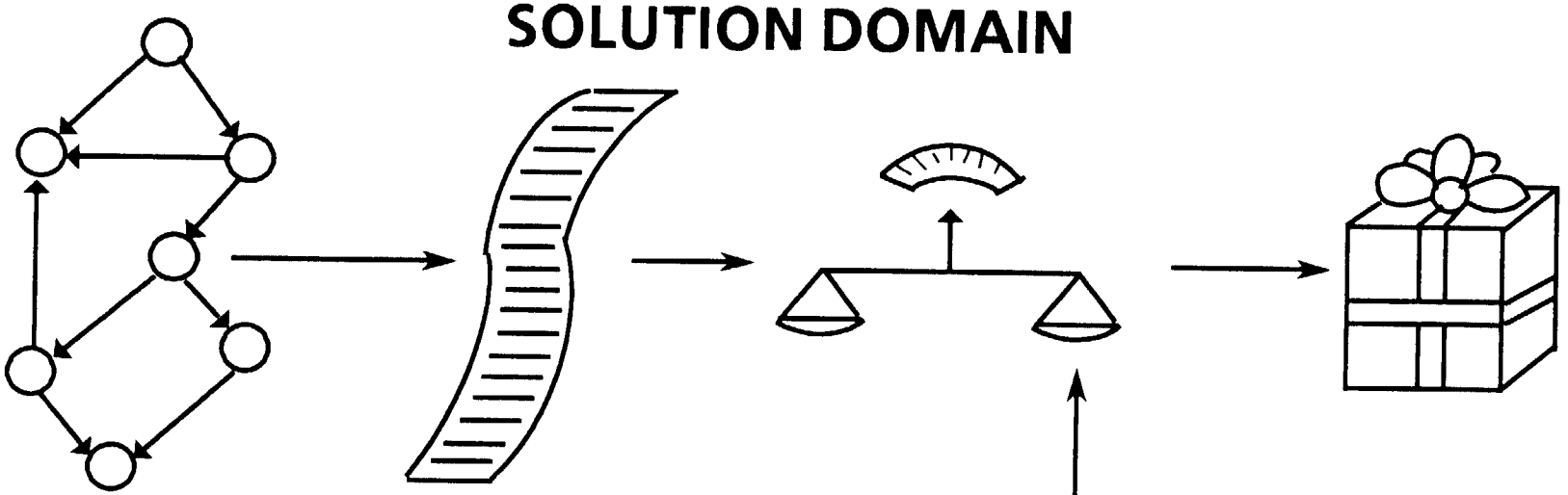
A PRACTICAL APPROACH TO OBJECT BASED REQUIREMENTS ANALYSIS

DANIEL W. DREW

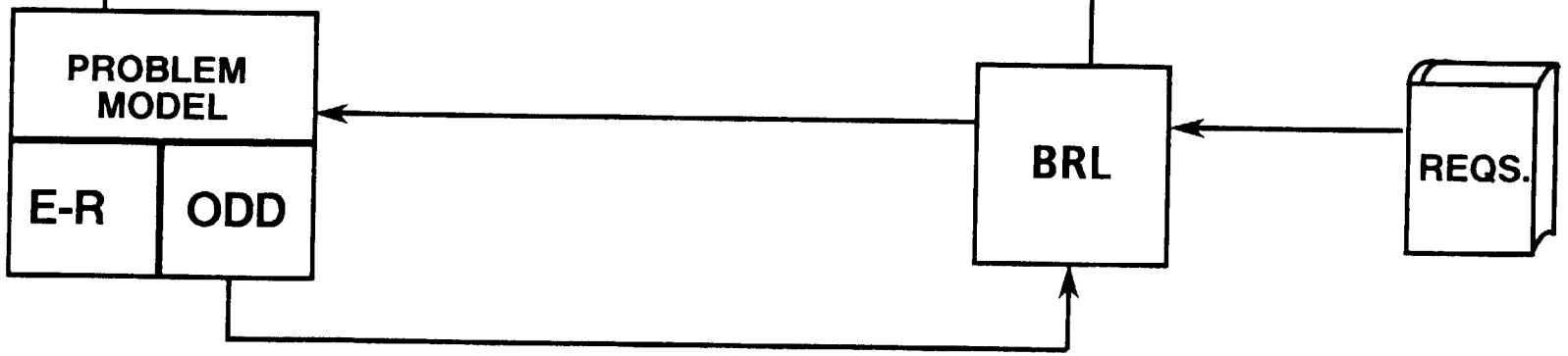


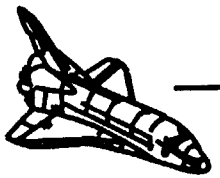
OVERVIEW OF APPROACH

SOLUTION DOMAIN

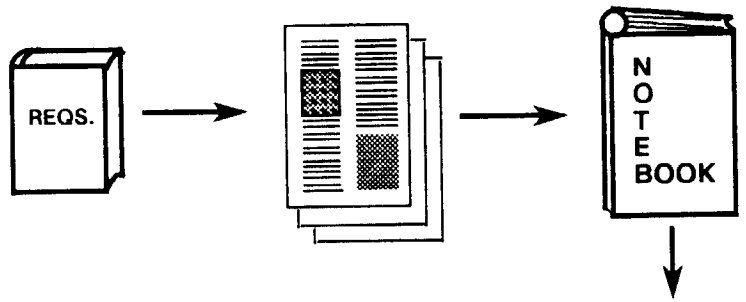


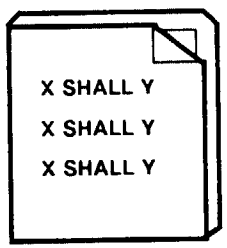
PROBLEM DOMAIN





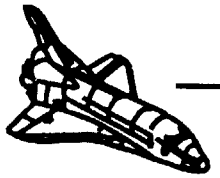
ORGANIZE REQUIREMENTS INFORMATION



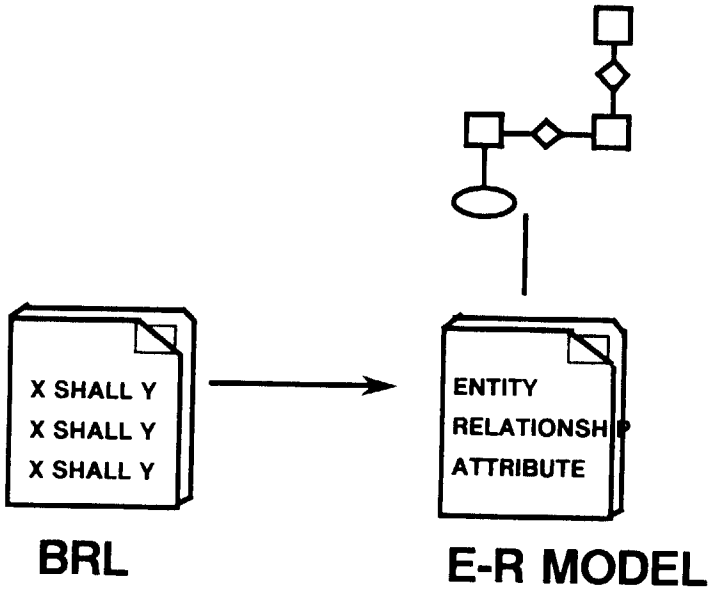


BRL

- AN ESSENTIAL FIRST STEP
- FORCES EXAMINATION OF EACH REQUIREMENT STATEMENT
- AUTOMATED TRACKING SYSTEM A MUST

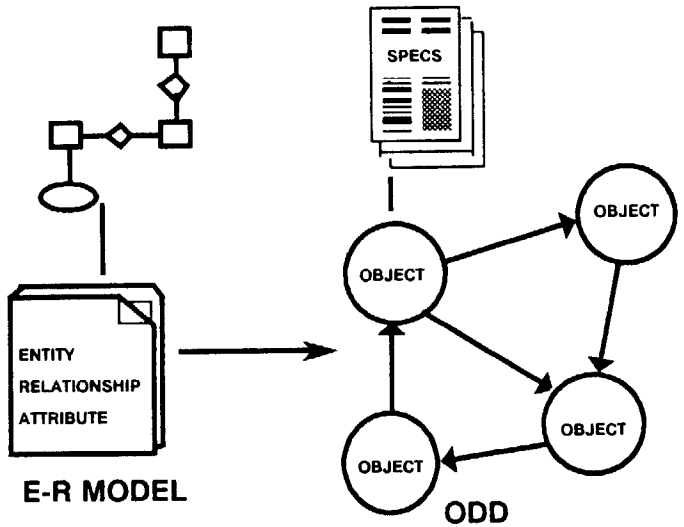


CREATE THE STATIC MODEL

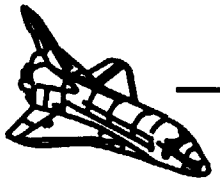


- STRAIGHTFORWARD EXTRACTION OF ENTITIES
- RELATIONSHIP/ATTRIBUTE IDENTIFICATION NOT AS CLEAR
- ENCOUNTERED DIFFICULTY IN ATTEMPT TO LEVEL E-R MODEL
- NEED FOR COMPUTER-AIDED TOOL

CREATE THE DYNAMIC MODEL



- SMOOTH TRANSITION TO OBJECTS
- NAMING OBJECTS VERY CRITICAL
- DIFFICULT TO AVOID DESIGN ISSUES
- MUST MAINTAIN DETAILED INFORMATION



SUMMARY

- PROBLEM IS TO REPRESENT INFORMATION
- USE DATABASE MODELING TECHNIQUES
- COMMUNICATION ISSUES MUST BE ADDRESSED
- CUSTOMERS EASILY UNDERSTAND OOD PRESENTATIONS

N91-10616

A Modernized PDL Approach for Ada Software Development

Paul Usavage Jr.
(215) 354-3165

M&DSO / Ada Core Team
Valley Forge, PA

ABSTRACT

The desire to integrate newly available, graphically—oriented CASE (Computer Aided Software Engineering) tools with existing software design approaches is changing the way PDL is used for large system development. In the approach documented here, Software Engineers use graphics tools to model the problem and to describe high level software design in diagrams. An Ada—based PDL is used to document low level design. Some results are provided along with an analysis for each of three smaller GE Ada development projects that utilized variations on this approach. Finally some considerations are identified for larger scale implementation.

BACKGROUND

In 1987, the Ada Core Team was formed within GE's Military & Data Systems Operation to apply advanced technologies including the Ada language to the development of large satellite ground systems that form our business base. GE M&DSO has been producing real time satellite ground stations for 15 years with a strong, established methodology. The addition of graphics workstations and graphics tools to this methodology is just a natural evolution of these methods. The techniques proposed here have grown out of GE's methodology and been refined through use on various Ada projects and IR&D work. The information in this paper is based primarily on the results of these efforts.

INTRODUCTION

The availability of automated graphic tools supporting structured analysis and structured design techniques, and the need for major improvements in productivity and quality are causing software organizations to rethink their software engineering methodologies. PDL (Program Design Language or Process Description Language) is the most commonly used design tool in many organizations. As a result there is a wide base of experience in PDL as a descriptive medium.

Yet, when an organization wants to add CASE (Computer Aided Software Engineering) tools to their existing methodology, it often is unclear what role PDL should play. Are PDL and graphic CASE tools redundant, or can they both contribute to modern software design practices? And what about the practice of coding some Ada constructs (notably package specifications) during detailed an even preliminary design? Does this narrow the scope of PDL's usefulness?

This paper is intended to document our analysis of the most effective tools for each portion of the software design cycle. Each tool, graphics, PDL, and Ada source code, has characteristics that make it useful to apply to part of the design problem. PDL has been used in the past for the representation of many design aspects. Today there are areas where PDL is best suited, and areas where other tools are better suited than PDL.

By way of further introduction, let us examine the traditional design approach and use of PDL.

TRADITIONAL APPROACH TO SOFTWARE DESIGN

Traditional documentation of a program with PDL involves two parts. The primary part is the process description, which is a description of the implementation or algorithm used in a program, subprogram, process, function or procedure. The second part is the prologue, which is usually present to support the process description by explaining input/output data items and local variables. The prologue often provides references to the design or requirements documentation, and usually includes information and format necessary to an automated PDL processor. Sometimes the term PDL is used to refer to just the process description, and others it is used to refer to the prologue as well. In this paper PDL will be used to refer either to the process description and to the language used for process description.

Software Design Phases

The evolution of a software design occurs in distinct steps over several project phases. During the **Software Requirements Analysis** phase, a software system is partitioned into Computer Software Configuration Items (CSCIs), and all software system requirements are allocated among these CSCIs.

During the **Preliminary Design** phase, a high level design is conceived for each CSCI sufficient to satisfy its allocated requirements. This design is described in English in a continuous, flowing, 'easy to read' paragraph format. Software hierarchy charts are usually prepared next for the design review. Database and file format designs are initiated during this phase to reflect attributes of the preliminary design.

The software design process continues during the **Detailed Design** phase with the generation of preliminary software source modules for each design component. The method to be used in these modules is described using a PDL process description. The first 'cut' at this description would likely be at a high level of abstraction (showing fewer details). Iterative refinements are then made of the PDL process description, assisted somewhat by the use of structure charts. The design is refined by adding more detail on how the module's functionality is to be provided. This lengthens the process description, and separate, subordinate modules are then created to break out cohesive elements of this process description. A PDL processor is used during this activity to check for syntax errors and to create calling trees and object/variable cross-references for analysis use.

The end of the PDL refinement process is reached when two criteria are felt to be satisfied. The first requires that the process descriptions should be detailed enough that the module can be coded by someone familiar with the technology but unfamiliar with the design. The second criteria requires that process descriptions must be of a suitable length (between 1 and 2 printed pages) to result in reasonably sized code modules. Consistency and quality are encouraged by the establishment of PDL standards, by the informal sharing of sample PDL, and by peer review or structured walk–through of the PDL processor printed output.

The **Coding** phase implements the design. The source code is written into the same modules already containing the prologues and PDL process descriptions. In some cases the source code is

interspersed throughout the PDL in a style that explains a step of conceptual processing with a block of PDL, then implements it with a block of source code. In other cases the entire process description is kept intact at the beginning of the module, followed by the entire source code. The former makes it easier to match PDL to source code, while the latter allows the PDL (and the source code) to be better seen and understood in whole.

Benefits Of Traditional Approach

Our Software Development section has enjoyed steady productivity gains since this PDL methodology was adopted. PDL usage has resulted in higher quality and greater productivity than previous development methods (which made use of, among other things, English prose descriptions and flowcharts). Of course, many factors are at work in increasing productivity including the availability of more and better hardware, but at least some of this improvement can be attributed to the use of a vigorous, robust, well-known and well-followed methodology. The use of PDL contributes to quality and productivity in the following ways:

1) Creation and maintenance of documentation is easier when employing the same tools (*e.g.*, computer terminals, editors) used in writing the source code.

2) Design descriptions are more complete, rigorous, detailed, and more standardized.

3) Design walkthroughs may be used more readily to reduce the number of design errors.

4) Some aspects of the design (*e.g.*, syntax, keyword balancing, call trees, indexing of references) may be checked automatically.

5) Deliverable documentation may be produced automatically from source code containing PDL.

6) Fewer errors are made when representing actual software implementation due to the proximity of PDL and source code.

7) Less effort must be spent on explanatory comments when the PDL is located with the source code.

Disadvantages Of Traditional Approach

Usage of this approach has also shown some disadvantages. Some of these are:

1) The 'easy to read' English prose used in preliminary design documentation is hard to write in a way that is free from ambiguity.

2) The PDL documentation for a large system is copious and very low—level in detail; it can be very difficult to find the PDL associated with a given aspect of system behavior.

3) PDL does not support well the more formalized *structured* approaches to partitioning (*e.g.*, analyzing coupling and cohesion) and automated checking, especially when experts try to review the partitioning decisions of others or when automated tools are used to *verify* the design.

4) PDL approaches traditionally have neglected the data part of a design

Advantages of Newer Graphic Tools

CASE tools now available automate graphically—oriented regimens in system analysis and software design. These tools include support for such approaches as Data Flow and Control Flow Diagrams, Structure Charts, Entity Relationship Diagrams, Object Dependency Diagrams, Object Interrelationship Diagrams, Data Dictionaries and integrated tool databases. GE has used the **team***work*® tool from Cadre Technologies, Inc. for the studies described in this paper.

The automated graphic tool approach to Structured Analysis and Structured Design has many commonly recognized benefits:

1) Communication via graphics seems to occur at a much higher information bandwidth, using visible relationships and psychological cues to more quickly attain a high level of reader understanding.

2) Graphics seem to provide better support in decomposing or partitioning a software problem or design, and in examining alternatives and reviewing the results.

3) Production of graphics for formal presentations and reviews is automated.

4) Tools can often assist in the storage, control of and access to information by design teams.

5) Tools can provide higher levels of automated balance and consistency checking by including a data dictionary, and in some cases can automate design verification.

6) Graphic tools seem to better represent system level behavior, interface design, and data design.

Disadvantages of Graphical Tools

Graphics CASE tools also have their disadvantages, including:

1) Graphics are generally less effective than PDL when dealing with larger quantities of low level details (for example, flow charts become considerably less attractive when used to document low level details of very large programs)

2) Newer, more complicated approaches may require much more extensive tool and methodology training to be successful.

3) Graphics CASE tools can involve a substantial additional investment in both hardware and software.

4) Development schedules must be adjusted to reflect additional time spent on the front-end design.

5) It is very difficult to prove (*e.g.*, to customer or business management) that the additional time and money spent up front results in cost savings later.

6) Human nature sometimes leads people to believe that the tool will do the work for you; really it just helps to represent work *you* do yourself.

PROPOSED METHODOLOGY

The following methodology, documented in our Software Development Plan, has been synthesised from our existing methodology and from proposals by many authorities. It has been adapted to complement our existing approach and is recommend by our group for GE's large development contracts. The phases here are much the same as in other approaches, including the classical waterfall approach and the default cycle documented in DoD—STD—2167A. Familiar activities occur during the phases but more effective tools, refinement techniques and documentation media are used.

The basic approach uses graphics at the higher levels of abstraction and PDL at lower levels. This documented approachsupports the use of the Ada language well. A non-Ada version of the Software Development Plan is planned to properly exploit this same methodology on non-Ada projects. The current Plan version makes use of object—oriented terms and methods. However, it is intended to support either object—oriented or functional decomposition of a system, or an approach that hybridizes the two.

Approach By Phase

The **Software Requirements Analysis** activity uses a basic Structured Analysis approach (as described by Yourdon & DeMarco, McMenamin & Palmer, Ward & Meller, Hatley & Pirbhai, and others) including the use of Data Flow and Control Flow Diagrams and a Data Dictionary for Essential and Incarnation models (see the references). The purpose of this is to model the problem in more detail in order to understand it. This is done first in a way that removes the consideration of technology from the statement of the problem solution, and then adds it back into consideration. The results of this analysis, in the form of Data Flow Diagrams, are input to the next phase of software development.

Preliminary Design involves the identification of Configuration Software Components (CSCs) from the Data Flow Diagrams. These may be high—level objects and operations identified in an Object—Oriented approach. Object Dependency Diagrams are produced for the identified objects. Interfaces between CSCs (and CSCIs if not done during Requirements Analysis) are defined, then depicted using package specifications. The package specifications are coded in Ada, showing the Ada declaration of each resource (mostly types and subprograms) exported from the package specification, along with Ada *with* clauses showing necessary dependencies. Compiling these interface specifications checks for consistency and makes a firmer foundation for further breakdown of development work. High—Level executive CSCs are described with PDL at this stage to show the major elements of control. The PDL for the executives would include the creation of their declarations in package specifications or as stand-alone subprograms, along with Ada with

clauses for their dependencies. The PDL consists of structured language process descriptions based on the Ada executable statements for iteration, loops, and conditionals. No attempt is made to compile the executives at this point, the purpose is to describe control dependencies inherent in the design. This PDL may in fact be contained solely within the CASE tool and not within a source code member at all. This makes it instantly accessible when documenting and refining later stages of the design.

The design process continues during the **Detailed Design** phase as structure charts are generated for each CSC. These show the architectural details involved in implementing the CSC. Computer Software Units (CSUs) are identified. These may be lower level objects in an object—oriented system. The implementation of individual CSUs are described in PDL process descriptions within the CASE tool graphics environment. This gives the programmer a better sense of partitioning and of the overall system structure than does writing the PDL into a disconnected source file. No compilation is attempted of these process descriptions. They are based on the Ada language syntax for universality of understanding, not for compilability at this stage. However, new interfaces derived at this detailed level of design (*i.e.*, more package specifications) are coded in Ada and checked with the compiler. These package specifications declare all types and data structures necessary to components external to the package specification. Also, within the package bodies, internal types and major internal data structures are coded in Ada and compiled. This helps to firm the data design and package dependencies. This is a major design component that is best described and checked with the Ada language and compiler itself.

The **Coding** phase that follows detailed design involves transfering the PDL from the CASE tool into existing and new Ada source modules, then writing Ada code for the design represented in the PDL process descriptions.

TRIAL PROJECTS

A number of GE Ada projects have been undertaken using variations on the traditional and proposed methodologies. The following projects have been selected to present some variety in approaches to PDL. No hard metrics are available for these projects to give insight into the contribution of methodology components, such as the number of errors created and found during a phase, or even created but not discovered. Instead, project team members were interviewed about problems, rework and errors that occurred. Their comments were then analyzed for apparent relation to the choice of methodology.

The projects described here are IR&D projects that have occurred over the last two years at GE. They appear here in chronological order, and in fact show an evolution in methodology over this time period. Methodology refinement was not the primary intention of these IR&Ds, each one was instead performed with what seemed the best approach to those directing the efforts at the time. Methodologies of later projects were of course tuned to benefit from the lessons of the earlier ones. Most participants were first time Ada programmers, although each project (after the first) had at least one person assisting during coding that had benefitted from some experience on a previous phase. The experienced people were not usually available during the design phase, however.

Project 1

One study in Ada software development involved the redesign and re-implementation of a predictive mathematical simulator. The project resulted in approximately 8000 compiled Ada statements (counted by semicolons, not including blank or commented lines). Automated CASE tools were not available during the study. Diagrams were produced using a PC-based general-purpose drawing tool. The Ada compiler itself was used to check the PDL for syntax. PDL consisted of coded and compiled Ada block constructs (e.g. loops, conditionals), compiled type and variable declarations, and Ada comments instead of procedural (sequential) statements.

During Preliminary Design, narrative English specifications were produced according to more traditional development methodology. Object/ Package Dependency Diagrams and Control Flow Diagrams were drawn. These were presented during the Preliminary Design Review (held at the end of the Preliminary Design Phase), but effort was not spent to maintain these diagrams for use during Detailed Design. High-level objects and procedures were identified and package specifications coded (but not compiled—the development environment was not available at the time).

During Detailed Design, the Ada package specifications were entered and compiled. Any interface errors detected then were corrected. Package bodies, subprograms and most types and

variables were declared in compiled Ada within the code modules.

In the Coding phase, the unimplemented (commented) portions of the compiled PDL bodies were coded and the components integrated and debugged.

The study was a quite a success as far as Ada software development was concerned. However, an analysis is possible of problems that arose during the study for possible effects of the choice of methodology. For instance, there was a wide variation among the six programmers participating in the study in the style and composition of the compiled Ada PDL. Some felt very comfortable during Detailed Design writing almost complete Ada code and very few PDL comments. Some felt very uncomfortable with the Ada syntax and compiler and wrote mostly comments and few compiled types/objects/block constructs. This sometimes resulted in inconsistent levels of abstraction of the PDL design description.

In general, the project tended to achieve different levels of abstraction and maturity at different times. It took longer for a programmer to write PDL that was mostly code. It took less time to write PDL that was mostly comments, but more time to write the source code in the next phase. Management misunderstandings resulted from this when attempting to assess the progress of the effort at a given point in time.

The problem with different styles of PDL and different PDL/code contents appears to be more common with projects that use an Ada compiler to check PDL. This also seems to occur more frequently when there is less experience with Ada and the PDL approach. One remedy for this is more and better training. Another is *not* to use the Ada compiler to check PDL syntax—and the problem goes away if a PDL processor is used which has a more forgiving syntax, or if only a visual check is performed on the PDL. The visual check is appropriate only if module sizes are kept small. After all, PDL syntax errors are only damaging if they cause ambiguity or incorrect interpretation in the design.

The problem with inconsistent levels of PDL abstraction that showed up on this project is common to many different approaches and processors. This is bad because it is confusing, it makes the design less understandable and less easily checked by others. Abstraction is useful because it hides those details unnecessary to this portion of the problem solution. The more localized the scope of detail, the less affected the system will be if it changes. Each person (or component of software) has to be an expert in fewer areas, and is free to concentrate and come up with a better, more pure solution in his/her/its own area. Removing unnecessary detail makes a system design more understandable, modifiable and robust.

The consistency problem decreases with programmer experience. Levels of abstraction can also be checked for consistency during peer review or structured walkthrough, giving feedback to the programmer and allowing the descriptions to be corrected. The best level of abstraction for a PDL process description of a given module is somewhere above (less detailed than) the level at which the source code for that module would need to be written.

Despite the apparent problems the team was able, however, to bring all portions of the system to completion by the end of the test phase. The productivity of the total effort was only very slightly lower (a few percent) than that of the more traditional projects. This was probably affected by a variety of factors including less effective training, lack of tools and technical difficulties with the platforms used, but also that slightly less documentation was produced than is normal.

Project 2

The second project for analysis was a 1988 IR&D effort to design and implement a platform-independent Ada binding for a Man-Machine Interface. Portions of the project made use of the graphic CASE tool when it was available. It used an Ada based, uncompiled PDL but no PDL processor. This project resulted in a larger design than was implemented, with about 2000 lines of compiled Ada code (again by semicolons, not including blank or commented lines) being produced.

During Requirements Analysis, Data Flow Diagrams were constructed to describe physical, logical, and incarnation models. The resultant diagrams were used during Preliminary Design to help identify high-level objects and to partition the system. Ada package specifications and their bodies were written (with subprograms deferred) and compiled to document the interfaces. Object Dependency Diagrams were drawn to show the object relationships.

During Detailed Design, extensive use was made of the Ada compiler. Drivers were identified and coded in Ada. Important type and object declarations were coded within the package bodies. A

key routine in each of the major objects/packages was coded and tested to ensure the feasibility of the design. A *key routine* was some subprogram that, when demonstrated, would validate most of the design decisions for the rest of the subprograms in an Ada package. Other, non-key subprogram bodies were designed and documented only in PDL within the source modules. This PDL used Ada syntax but was commented and not compiled. Some type and data declarations were coded compiled. Some structured design diagrams were constructed but not many. The burden of design documentation and analysis and refinement was performed using compiled package specifications, compiled *key routines*, and PDLed subprograms. The CASE tool was not continually available during this phase due activities involving the tool evaluation and purchasing mechanism.

During the Coding phase the subprograms already expressed in PDL were expanded to code. The coded portion of the system was integrated, tested and demonstrated.

Again, the overall project was successful but some useful methodological refinements may be suggested from observation. One such observation is that because the graphic CASE tool was not always available during the project, a graphics approach was not taken during much of the preliminary and detailed design stages. Instead, emphasis was placed very early on representing the design with coding package specifications and bodies. Much rework was involved as new alternative designs were identified, coded in Ada package specifications and bodies, reviewed, then modified. The normally constructive and necessarily iterative process of conceiving a solution, expressing it, evaluating it, and suggesting other alternatives suddenly seemed to involve too much effort and be too destructive to the participants.

One possible approach to this difficulty of rework involves exploring the design in more detail, using graphics and PDL within the CASE tool, before package specifications are coded. The tool has fairly good support for this. Balancing is checked, and creation and modification of graphics is made easy within a window—and—mouse oriented environment. The tool checks balancing and graphic relationship rules for the resulting diagrams. Then, when the Ada package specifications are coded and compiled, they are built on a foundation of previous work which has already involved consideration of many of the possible alternatives. There should be less need for generating alternatives.

Overall, the productivity of this project met that of other projects in our organization's past.

Project 3

The third project was the most recent and the most closely matched to the proposed methodology. The late—1988 project completed the coding and testing phase during the writing of this paper. It redesigned and coded two CSCs (functions) of a prototype real-time distributed ground system in Ada. Over 7000 lines of Ada code (measured by the same criteria as in the other projects) were written. Extensive use of the graphic CASE tool was made throughout the entire design effort. Again, an automated PDL processor was not used.

During the Software Requirements Analysis phase, the system was modeled in Data Flow Diagrams. During Preliminary Design, these DFDs were used to generate Objects and Operations, and Object Interrelationship Diagrams were drawn using the CASE tool. Major objects were coded as Ada package specifications, with their operations being the subprograms exported from the package specification.

During Detailed Design, Structure Charts were drawn showing the interrelationships of each objects operations in performing some component of the system's purpose. Each operation was described with Ada—based PDL within the confines of the CASE tool. Refinement was performed by editing the PDL to increase the detail, then breaking out pieces of this new detail into new software components and creating new modules for them in the structure chart. When analysis and review of the structure charts and PDL met with satisfactory results, matching Ada package specs were created. Each specification was coded to show the exported resource (mostly types and subprograms) and the procedures stubbed out. PDL prologues were placed in the Ada modules, but no PDL. The PDL remained within the CASE tool database retrievable through the structure charts.

During the Coding phase, the subprograms were written in Ada either from the PDL printed from the CASE tool, or from the same PDL cut and pasted into the modules through the window and mouse-oriented workstation environment. The design information remained available within the CASE tool database (and would be delivered that way, in a soft copy documentation scheme for deliverable software).

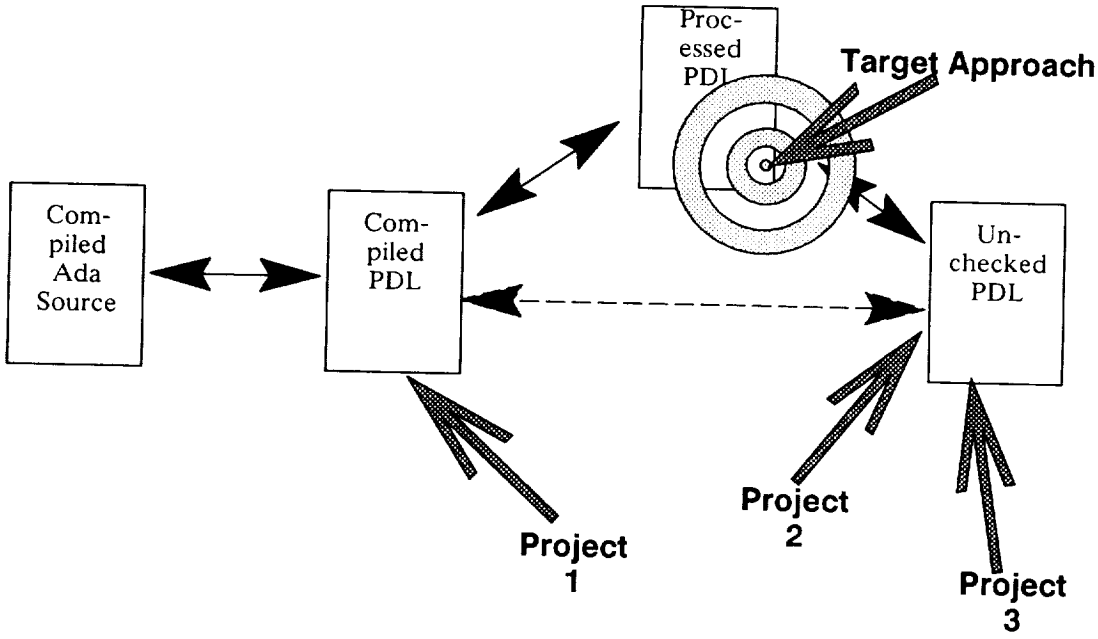
This approach seems to have paid off in a number of ways. Partitioning seems to have been so fully explored using the CASE tool that little rework of

compiled Ada package specifications was necessary. Design alternatives were efficiently analyzed within the CASE tool, where graphic and PDL information combined to give a good view of the system at several different levels of abstraction.

Module sizes were judged to be excellent: a half page maximum of PDL. Quite a few modules tested correctly when first compiled, even when coded from PDL by a first–time Ada programmer. This was attributed to the simplicity of the modules and the clarity of the PDL, which in itself might be attributed to the quality of partitioning.

The quality of the PDL seemed to be enhanced by its proximity to the graphic representation of the overall hierarchy, and the relative ease of traversal from PDL description to PDL description throughout the hierarchy. This ease of use contributed to good partitioning showing good coupling and cohesion characteristics.

The productivity on this project seems to be well ahead of that established for traditional projects (in the ball park of a 10–20% improvement for a first Ada project).



A view of PDL alternatives and our target approach
Figure 1

CONCLUSIONS AND SUGGESTIONS

Choice of Representation

One general theme in the methodology is to explore a design fully given the tool appropriate to the level of abstraction. The choice of tool should efficiently allow representation of that level of abstraction, and allow review, generation of alternatives, and easy representation of the final choice. Alternatives should be explored fully and adequately at the design stage under consideration, with the tool that does so in a most efficient (and reliable) manner.

Graphics seem to be a useful, powerful, and efficient tool for upper to middle level design. They also, with the proper tool, serve as an outstanding mechanism for indexing or gaining access to the low level of design. A graphical tree structure with a system breakdown is more easily understandible and more efficient a representation when searching for a given piece of a system than anything that we've seen before.

Quality and Testing

The alternatives and final choice of design from a phase should be subjected to *some form of testing*, that is, analysis, review, compilation, balance checking, or whatever else can be done to find as many errors as possible and to demonstrate as much quality as can be demonstrated. This provides a firmer foundation for the work that follows in development. As everyone knows, latent (un-

discovered) errors output from a phase are much more expensive to fix in later stages.

Scaling Up to Large Systems

The methodology was designed from experience in large systems—for application on large systems. The one place where scaling will change emphasis is on the choice of and number of tools. No PDL processor was used at all for any of the examined projects. This was due to the size of the projects versus the cost of procuring a tool. This approach should be re-examined for a larger projects.

On larger projects with more people it is more difficult and more important to have consistent, quality PDL. A PDL processor can contribute toward this goal. It certainly doesn't hurt to automatically check PDL for syntax and balancing errors, as long as the correction of errors does not detract from the creativity of design as sometimes happens with a strict Ada compiled PDL. No PDL processor is currently available that is integrated with the chosen CASE tool, but alternatives are being evaluated.

THE VIEWGRAPH MATERIALS

FOR THE

P. USAVAGE, JR PRESENTATION FOLLOW

A Modernized PDL Approach for Ada Software Development

Paul Usavage, Jr.
Ada Core Team

Ground Systems Department
GE/Military & Data Systems Operations

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Agenda

Introduction

Traditional Approach

Proposed Approach

Three Study Projects:

- Methodology
- Analysis

Conclusions and Suggestions

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Introduction

The Problem:

Upgrade existing strong, large–system methodology to:

- add the benefits of new graphics–based design tools and methods
- produce DoD—STD—2167A design documentation
- incorporate the benefits of the Ada language

The Investigation:

- graphic CASE tools
- graphical methods for Structured Analysis/Structured Design
- automated document production
- high–performance workstations
- Ada–based PDL language
- compiled Ada package specifications

The *New* Problem:

Which tool *best fits* which part of the design process?

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Proposed Improvements

Requirements Analysis Phase:

- Software design based on accumulated results of Structured Analysis
- Requirements depicted using Data Flow Diagrams and other graphical representations
  - ☞ helps to understand the problem when decomposing system

Preliminary and Detailed Design Phases:

- Design represented with integrated graphics and PDL
- Edit PDL from CASE tool windows within graphic structure chart
- Interfaces represented in compiled Ada code
- Iterative refinement performed against graphics and PDL together
- Preliminary & As–Built design documents automatically produced
- Graphics high level design diagram serves as index to PDL
  - ☞ ties together small PDL descriptions into *big design picture*
- Tool maintains design database for shared use
- Innovative approach with necessary stability for large systems

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

First Study Project – Background

Project:

- 8000 Ada statements – Mathematical Simulator

Methods:

- High–level design and requirements in English prose
- Structure charts used for presentation
- Object–oriented approach
- Coded high–level interfaces as Ada package specifications
- Used compiled Ada PDL
- Compiled block constructs—loops, conditionals
- Ada comments for sequential statements

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 1 — Analysis

Limited use of graphics, mostly for presentation purposes

- Graphics constructed with PC–type drawing tool, separately from system where code was developed
- Graphics were not accessible:
  - ☞ not effective partitioning analysis tool

Partitioning analysis performed with compiled PDL

- Compiler detection of syntax errors detracted somewhat from effort spent on design representation
- Extra effort involved for some people to represent design in compiled Ada
- Made it more difficult to represent alternative designs and decide between them
- Lower level design representation could obscure higher level tradeoffs

Design alternatives took more time to work out than if quicker representation were available

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 2 – Background

Project:

- 2000 Ada statements
- Designed and implemented Ada Binding for Man–Machine Interface

Methods:

- Structured Analysis generated Data Flow Diagrams (DFDs)
- DFDs used to partition system into components
- Object–Oriented design used for system partitioning
- Ada package specifications used to show object interfaces
- Object Dependency Diagrams [Booch] used to analyze objects
- Structure Charts and Buhr Diagrams used for presentations but not for partitioning decisions
- PDL stored with Ada source code
- PDL processor not used

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 2 – Analysis

Analysis:

- High–level partitioning performed based on Data Flow Diagrams
- Implementation–level partitioning performed using PDL and compiled Ada package specifications
- Structure Charts used for presentations, not for partitioning analysis
- Partitioning with code and PDL took longer to evaluate alternatives, and more work to incorporate changes
- PDL stored with Ada source code
- CASE tool was not available to store PDL within graphic notation
- It was somewhat tedious to find a given passage of PDL if you were not intimately familiar with the design
- PDL processor not used – no automated index production

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 3 — Background

SOFTWARE DEVELOPMENT

Project:

- 7000 Ada statements
- Redesigned and implemented in Ada 2 functions of Real-Time Distributed Ground Station prototype

Methods:

- Structured Analysis generated Data Flow Diagrams
- DFDs used to partition system into components
- Object-Oriented design used for system partitioning
- Ada package specifications used to show object interfaces
- Object Dependency Diagrams [Booch] used to analyze object partitioning
- PDL stored within each process box in Structure Chart
- Structure Charts and PDL used to repeatedly analyze and refine software partitioning
- PDL processor not used

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 3 — Analysis

Analysis:

- High level partitioning begun using Data Flow Diagrams
- Object–Oriented approach used to go from Data Flow Diagrams to an Object—Oriented Design
- Structure Charts and PDL used to repeatedly analyze and refine software partitioning
  - ☞ Partitioning and analysis with structure charts and PDL within the same tool allowed more thorough analysis and refinement of software partitioning
- Graphics seem to be very effective for analyses of system structure

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Conclusions and Suggestions

SOFTWARE DEVELOPMENT

Representation:

- Explore design fully with the best tool for that level of abstraction
  - ☞ Graphics are best for upper and middle level design
  - ☞ Graphical tree–shaped system breakdown for entire system (top to PDL) is most effective
- Examine partitioning alternatives, review, decide on the best approach before going on to next phase/abstraction/representation

Quality and Testing:

- All results and choices should be tested/analyzed/reviewed/checked for errors before building next layer of design

Scaling up:

- PDL processor (somewhat forgiving of syntax) is helpful to find true design errors
  - ☞ should integrate with graphics CASE tool environment

Representing Object Oriented Specifications and Designs with Extended Data Flow Notations

Jon Franklin Buser
Paul T Ward

Abstract

This paper addresses the issue of using extended data flow notations to document object oriented designs and specifications. Extended data flow notations, for the purposes of this paper, refer to notations that are based on the rules of Yourdon / DeMarco data flow analysis. The extensions include additional notation for representing real-time systems as well as some proposed extensions specific to object oriented development. The paper will state some advantages of data flow notations, investigate how data flow diagrams are used to represent software objects, point out some problem areas with regard to using data flow notations for object oriented development, and propose some initial solutions to these problems.

Introduction

Data flow diagramming is a general graphic-based modeling notation that has gained wide industry acceptance as a software specification and design tool. The proponents of object oriented techniques claim that systems built using these techniques have a natural system architecture that allows easier system modification and software component reuse. The authors support a method of system building that follows an object oriented development strategy and uses extended data flow notations to document the specification and design. There are many reasons for using data flow notation as the documentation medium:

- The notation is supported by a large number of Computer Aided Software Engineering (CASE) tools.
- Data flow models are not specific to any particular computer language, operating system, or hardware configuration making the necessary investment in training and tools useful over a wide spectrum of projects.
- Data flow modeling has a relatively long and successful record within the computer industry; many software engineers already have a working understanding of the notation.

Data flow diagrams use circles to represent processes, or units of work within a system, and arrows to represent data that is supplied to and produced by the processes [1]. Data Flow diagrams can be used for modeling general problem domains. These domain models are then evolved into software system specifications and designs. Figure 1 is a data flow diagram describing a Data Storage and Reporting System. The system produces reports on stored data and has a menu driven user interface for adding and updating records. A complete specification for the system would also include a detailed description of each process explaining how it will produce its output given the input data supplied. The Ward / Mellor[2] and Boeing / Hatley [3] real-time extensions introduce additional graphic symbols that are used to integrate finite state machine logic into the model. These state machine models strictly define the relationship of operations within a model and can potentially be executed to demonstrate the correctness of the model.

Object Oriented Partitioning

One of the key features of a data flow model is that it may be partitioned and leveled. This means that a number of processes can be grouped together into a single higher level process that represents the combined operations of the lower level processes. The highest level diagram in the model (the context diagram) represents the system as a single process and uses rectangular boxes to represent entities that are external to, but interact with, the system being modeled. Figure 2 is a

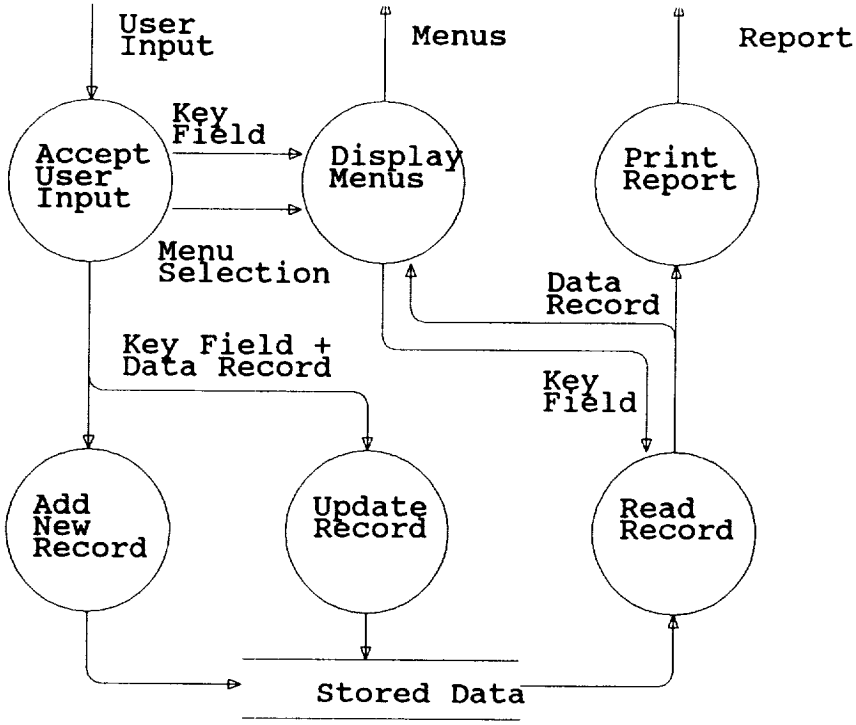


Figure 1

context diagram for the Data Storage and Reporting System.

Traditionally, data flow models have been partitioned by using a strategy called functional decomposition. This is a top down method that identifies high level system functions and then details, at the next level of the model, what processes will be required to perform each function. This process is repeated until all of the system's primitive components have been identified. Figure 3 shows a possible functional partitioning of the Data Storage and Reporting System. The system is partitioned into two sub-systems: one for managing data input and the other for data reporting. Both sub-systems have direct access to the data store.

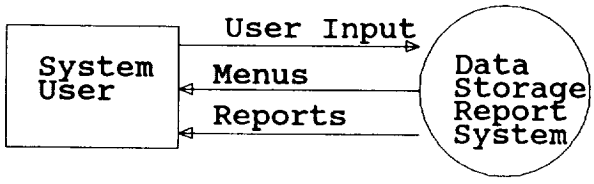


Figure 2

There are other partitioning methods. One alternate strategy groups together processes that are parts of the response to a given external event. Another organizes the model so that the number of data flows between the higher level processes are minimized. The choice of system partitioning is important because it will define the major subsystem interfaces and, in the case of large software projects, it will probably define the management structure of the organization that builds the system.

Object oriented specifications are produced by changing the criteria used when partitioning the model. With the help of information modeling techniques, classes of real world objects are identified in the problem domain [4]. Then the data flow model is partitioned by grouping together the processes associated with each object or class. In the case of the Data Storage and Reporting System we will identify a user interface object, a report object, and a data store object. These specification objects may be useable directly as

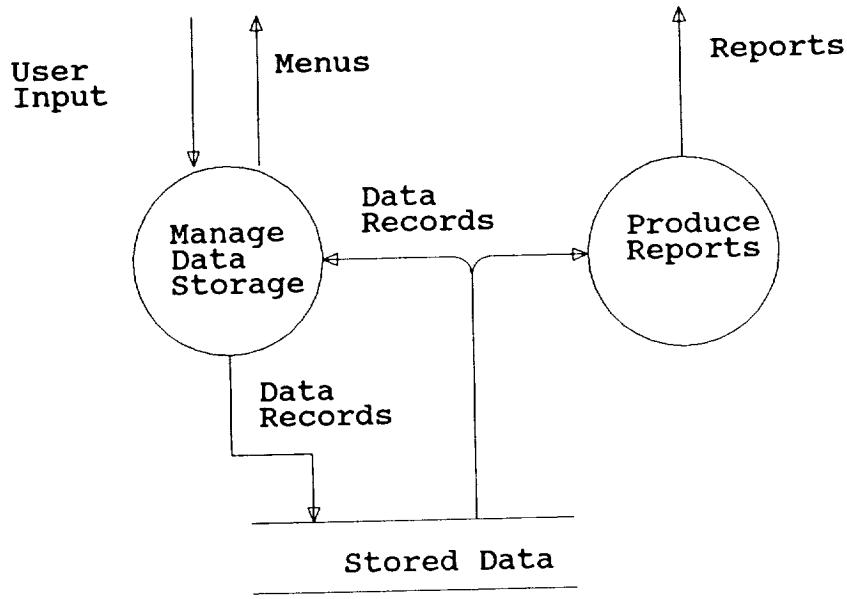


Figure 3

design objects, or they may have to be modified to transform them into design objects (e.g., to meet system performance constraints). These design objects can then be implemented as information hiding modules or Ada packages.

Data Flow Problems

We have found the object oriented partitioning strategy useful, however some of the rules governing traditional data flow diagrams and the CASE tool implementations of these rules conflict with object oriented goals.

One goal of object oriented design methods is to identify reusable objects. These objects may be reused within the same model or in different but related problem domains. Many of the CASE tools have a problem with regard to reusing these objects in the same model because the CASE tools typically enforce that all processes have unique names. If we want a process to be reused within a single model, naming conventions have to be devised to specify that different instances of the process are really the same. Of course, without additional tool support it is impossible to prevent different instances of each object from being modified so that they are no longer the same.

Another problem is that objects designed with reuse in mind will often be built in a more general manner than ones that have been engineered for a specific use. The result of this is that all of the object's access functions or methods may not used in a specific instance of the object. One of the primary model validation criteria applied to data flow diagrams is that all of the input and output flows entering a process must exist in the lower level description of the process. The existing CASE tools will report errors when general reusable objects are used in a model that does not make use of all the object's capabilities. For example, a

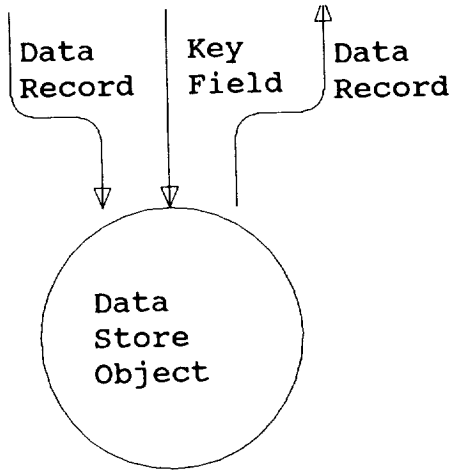


Figure 4

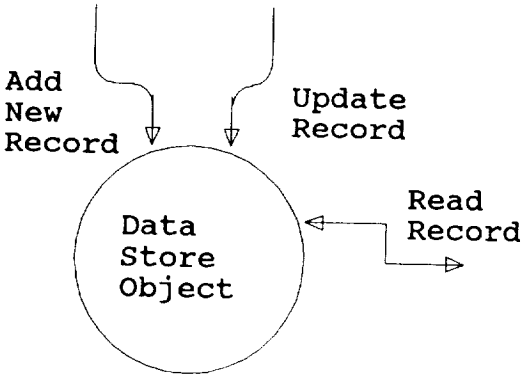


Figure 5

Add New Record =

Input Key + Input Data Record

Update Record =

Input Key + Input Data Record

Read Record =

Input Key + Output Data Record

Figure 6

more general data store object for the Data Storage and Reporting System might have a process for deleting records from the store. If this object is instantiated in an application that does not require a delete capability the analysis routines in the current CASE tools will report an error. To successfully level-balance the model, the delete process and its associated flows will have to be removed. A CASE tool designed to support importation of reusable objects must have a facility for deactivating specific access routines.

Representing Access Functions

Data flow models can be partitioned so that processes are grouped together in an object oriented fashion. The rules of data flow notation also allow data flows to be grouped together. This is commonly done to reduce the clutter of data flows entering and leaving higher level processes. We propose that the data flows should be grouped together so that all of the input and output parameters of each access routine are combined,

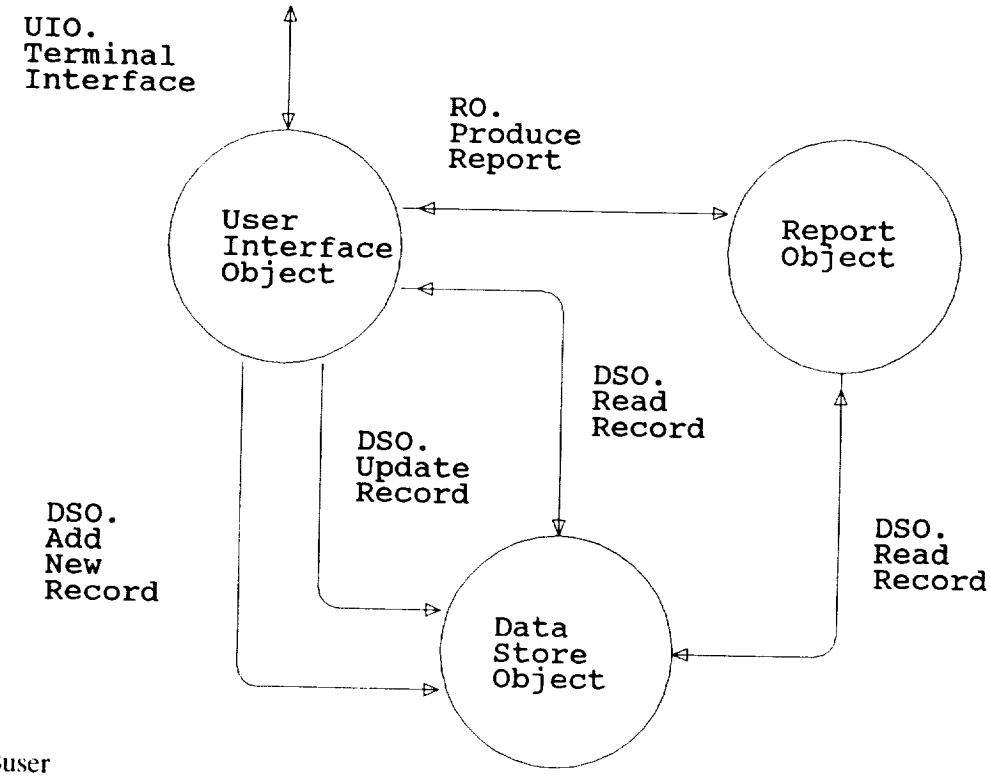


Figure 7

and that the combined flow is named for the access routine that it represents. If this approach is not followed it is impossible to determine which data flows operate together. Figure 4 shows the data store object from the Data Storage and Reporting System. Notice that all information that correlates input and output data with specific object capabilities has been lost. Compare this to figure 5 which groups the object's input and output flows together according to which access routine they are associated with. Information about the object's access routines is now retained. Figure 6 shows the composition of the each of the flows from figure 5.

Some CASE tools allow a data flow to have arrows on both ends indicating a two way flow of information. We suggest that this is a useful convention for representing flows that have both an input and output component. This notation is not completely adequate though, because it will not be clear from this diagram which object is using the other. This problem could be alleviated by introducing a new graphic symbol to indicate the direction of these combined flows or by applying naming conventions. One naming convention could name the flow by concatenating the objects name with the access function name, another convention could specify whether a particular flow component was an input or output (e.g., "input data record" as opposed to just "data record"). Figure 7 shows how the data store object integrates with the rest of the Data Storage and Reporting System using the double arrow head convention.

Future Work

Data flow diagrams can be used to model object oriented specifications and designs, however additional conventions may be needed for this to work well. Further work is needed to identify all of these conventions and to integrate them into CASE tools. Two areas of particular need are tools that will support the concept of inheritance, and browsers that can scan reusable software object libraries documented with data flow diagrams.

References

[1] T. DeMarco, *Structured Analysis and System Specification,* New Jersy: Prentice-Hall, 1978

[2] P.Ward and S. Mellor, *Structured Analysis for Real-Time Systems*, New Jersy: Prentice-Hall, 1985.

[3] D. Hatley and E. Pirbhai, *Strategies for Real-Time System Specifications*, New York: Dorset House, 1987.

[4] S. Mellor and S. Shlaer, *Object Oriented System Analysis*, New Jersy: Prentice-Hall, 1988.

THE VIEWGRAPH MATERIALS

FOR THE

J. F. BUSER PRESENTATION FOLLOW

Representing Object Oriented

Specifications and Designs

with

Extended Data Flow Notations

by

Jon Franklin Buser

Paul T Ward

Software Development Concepts Background Information

- **Real-Time Data Flow Diagram Extensions**
- **Develop Courses and Teach Real-Time Specification and Design Methods**
- **Work with CASE vendors**
- **Continued Research into Real-Time Development and Object-Oriented Methods**

Goal

Develop ways to represent object oriented designs and specifications with Data Flow Diagram based notations.

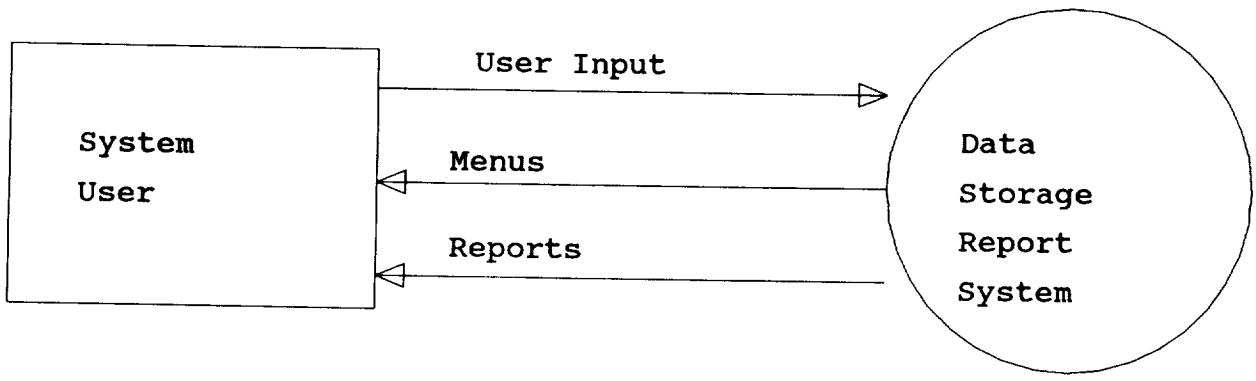
Advantages of Data Flow Diagrams

- Supported by many CASE tools
- NOT specific to any computer language or operating system
- Many Software Engineers already have a working understanding

Data Flow Problems

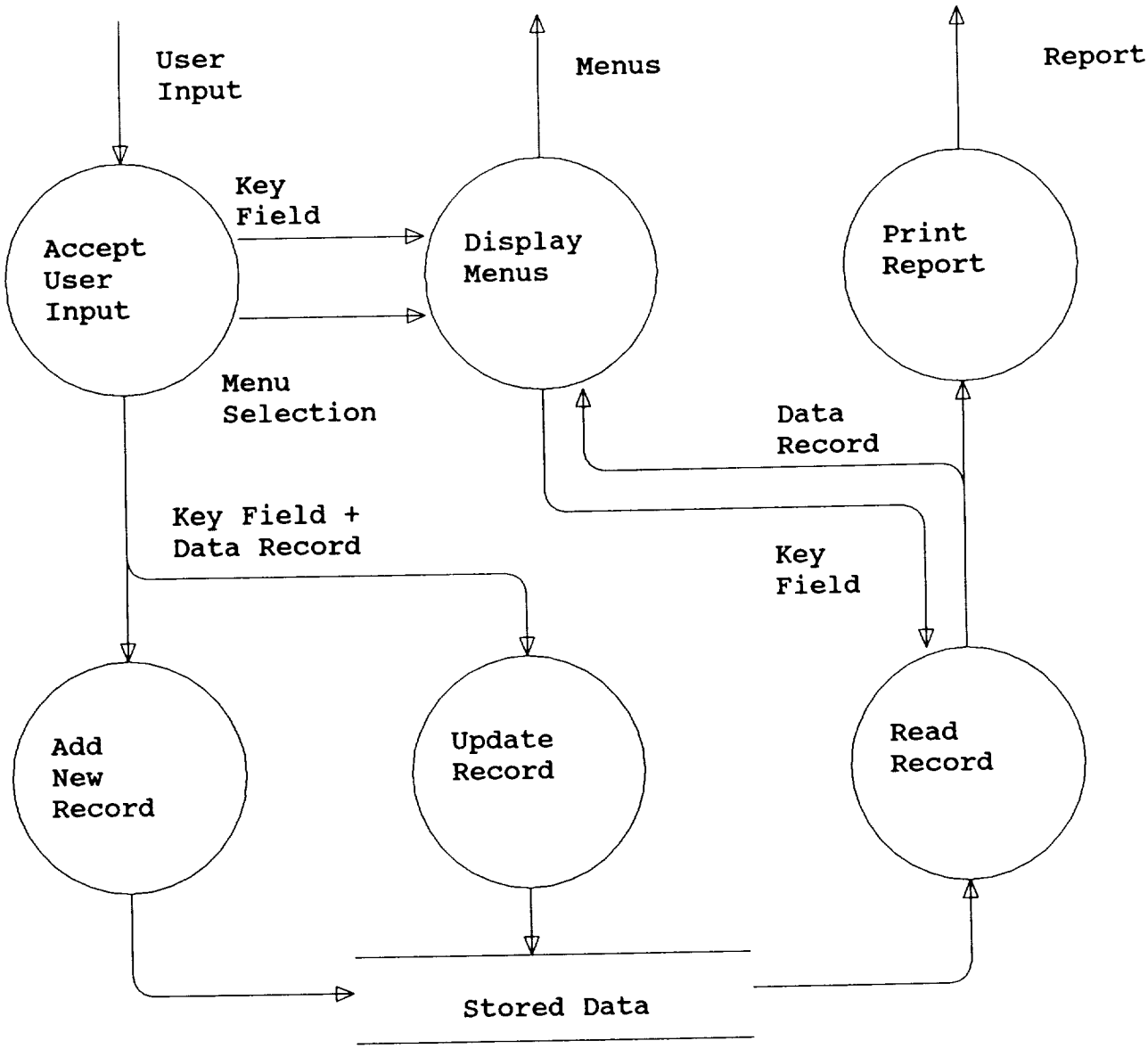
- CASE tool enforced unique names conflict with component reuse
- Level-Balancing conflicts with building general reusable components that have unused access functions
- Commonly used partitioning strategies do NOT reinforce the concept of Software Objects

The Data Storage and Reporting System



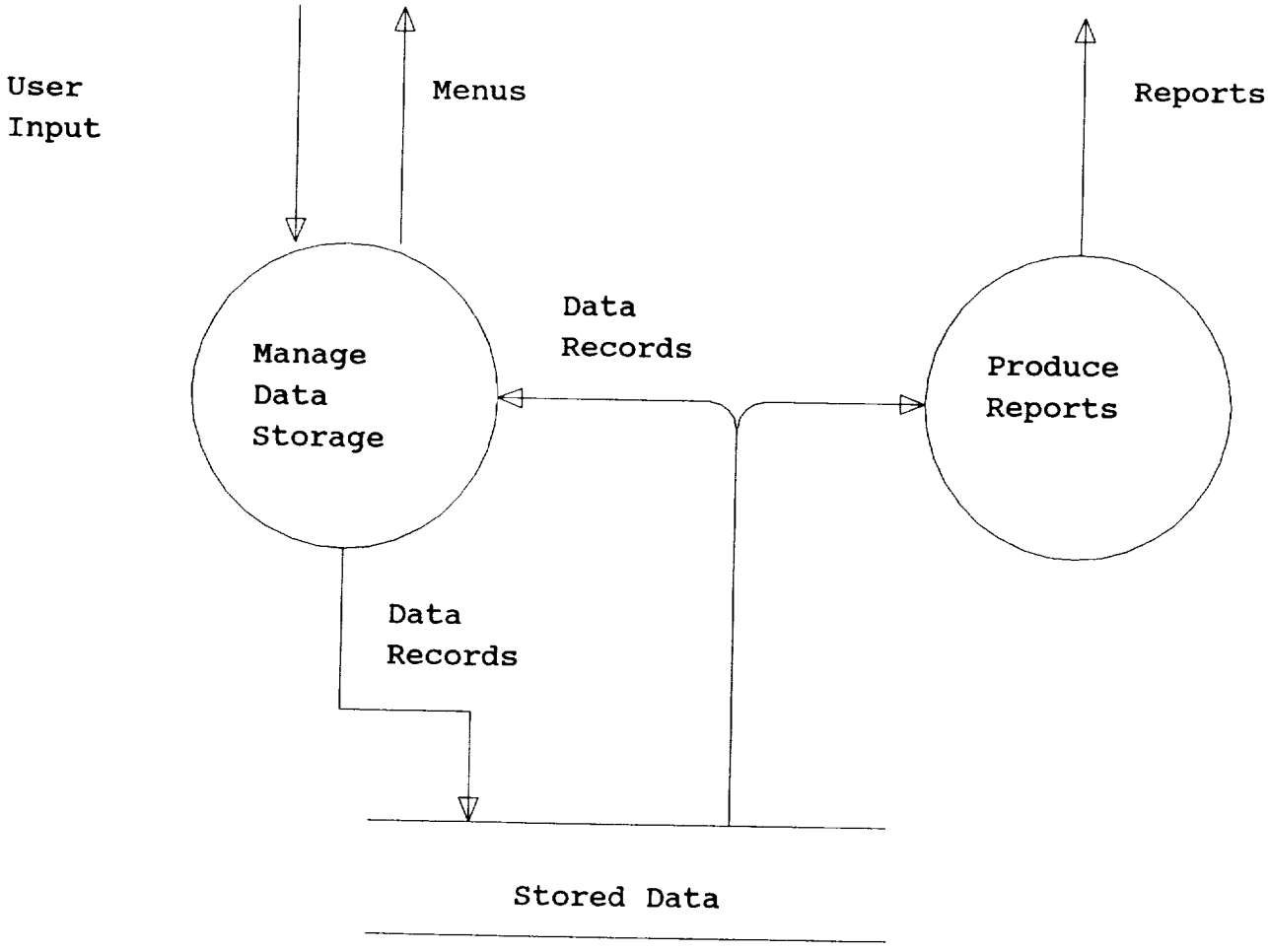
Data Storage and Reporting System

Detailed View



Data Storage and Reporting System

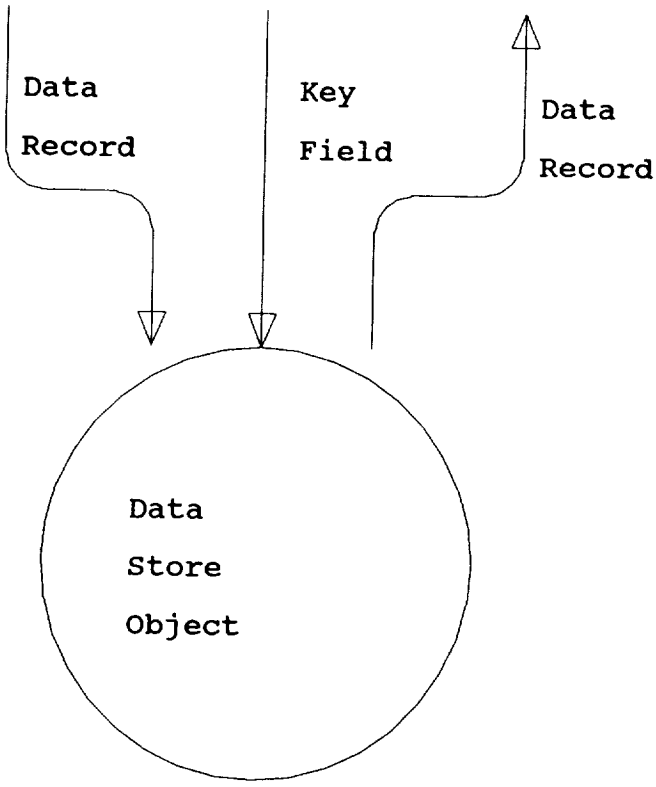
Functional Partitioning



Objects in the Data Storage and Reporting System

- Data Store Object
- Report Object
- User Interface Object

The Data Store Objects grouped together

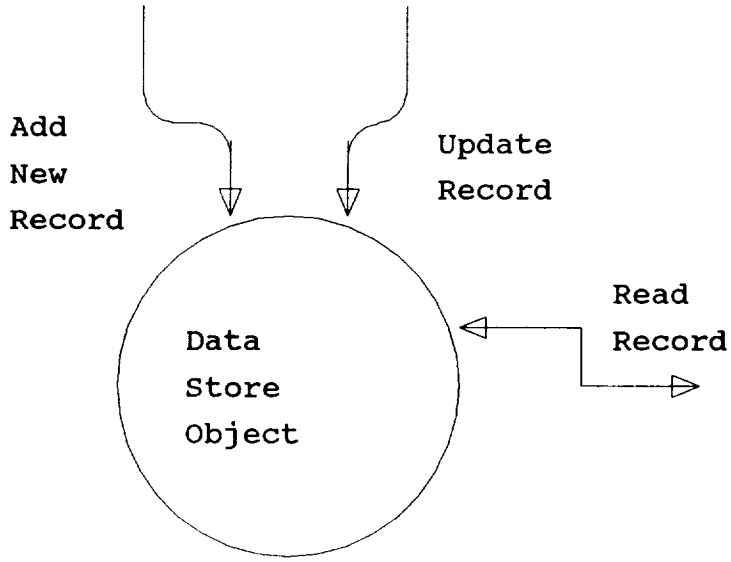


New Partitioning Conventions

for Representing Objects

- Group together processes that operate on the same real-world objects
- Group together Data Flows that are associated with the same process or access routine
- Name the combined flow for the access routine that it is attached to
- Use double arrow head if the flow is composed of both input and output flows

The Data Store Object



Add New Record =

Input Key + Input Data Record

Update Record =

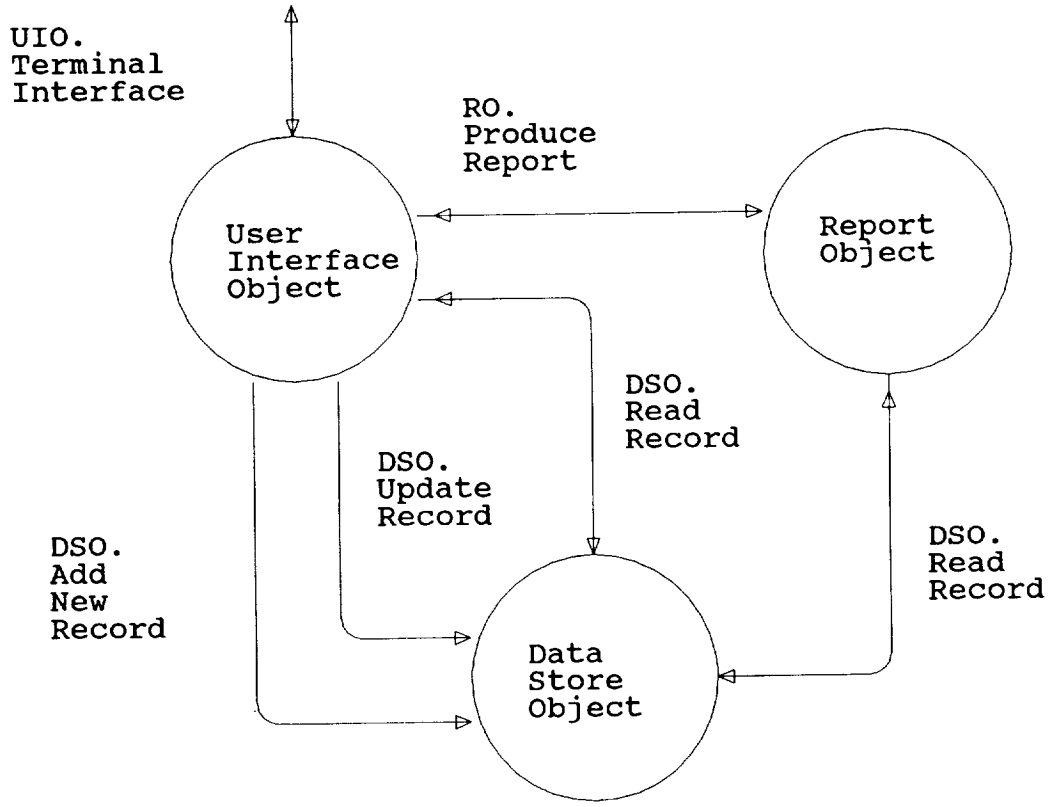
Input Key + Input Data Record

Read Record =

Input Key + Output Data Record

Object Oriented View of the

Data Storage and Reporting System



Future Work

- Work further with these conventions
- CASE tools to support reuse and inheritance
- Browsers to scan libraries of reusable components documented with Data Flow Diagrams

APPENDIX A

ATTENDEES OF THE 1988 SOFTWARE ENGINEEING WORKSHOP

ATTENDEES OF THE 1988 SOFTWARE ENGINEERING WORKSHOP

ADLER, DAVID	THE MITRE CORP.
ADLER, JONATHAN	UNIVERSITY OF MARYLAND
AGRESTI, BILL	MITRE CORP.
AMMANN, PAUL	THE SOFTWARE PRODUCTIVITY CONSORTIUM
AMSLER, JOHN	OAO CORP.
ANDERSON, MARSHALL	DEPT. OF DEFENSE
ANGIER, BRUCE	INSTITUTE FOR DEFENSE ANALYSIS
ANTONOPULOS, BETH	NASA/GSFC
ASTILL, PATRICIA	SIGMA DATA SERVICES
AUSTIN, GIL	IIT RESEARCH INSTITUTE
AZZOLINI, JOHN	NASA/GSFC
BARBER, GARY	INTERMETRICS, INC.
BARKSDALE, JOSEPH	NASA/GSFC
BASILI, VIC	UNIVERSITY OF MARYLAND
BAYNES, PERCY	VITRO CORP.
BEALL, DANIEL	FORD AEROSPACE CO.
BEARD, R	
BEARDSLEY, KARLA	THE MITRE CORP.
BECK, HANK	JET PROPULSION LAB
BEIERSCHMITT, MICHAEL	FORD AEROSPACE
BENNETT, TOBY	FORD AEROSPACE CORP.
BIGWOOD, DOUGLAS	LOCKHEED CORP.
BISIGNANI, MARGARET	THE MITRE CORP.
BLAGMON, LOWELL	NAVAL CENTER FOR COST ANALYSIS
BLAND, SKIP	UNISYS CORP.
BLUM, BRUCE	THE JOHNS HOPKINS UNIVERSITY
BODIN, JOSEPH	COMPUTER SCIENCES CORP.
BOND, JACK	NATIONAL SECURITY AGENCY
BOOTH, ERIC	COMPUTER SCIENCES CORP.
BOYCE, MARY-ANN	RMS TECHNOLOGIES
BRANCH, EDWARD	DEPT. OF DEFENSE
BREDESON, MIMI	SPACE TELESCOPE SCIENCE INST.
BREDESON, RICHARD	OMITRON
BRILLIANT, SUSAN	UNIVERSITY OF RICHMOND
BRINKER, ELISABETH	NASA/GSFC
BROPHY, CAROLYN	UNIVERSITY OF MARYLAND
BROWN, DAVID	AUBURN UNIVERSITY
BROWN, JAMES	JET PROPULSION LAB
BUCHANAN, GEORGE	IIT RESEARCH INSTITUTE
BUELL, JOHN	COMPUTER SCIENCES CORP.
BURCAK, THOMAS	PLANNING RESEARCH CORP.
BURLEY, RICK	NASA/GSFC
BUSER, JON	SOFTWARE DEVELOPMENT CONCEPTS
BUTSCHKY, MICHAEL	COMPUTER SCIENCES CORP.
CALDIERA, GIANLUIGI	ITALSIEL SPA
CARMODY, CORA	PLANNING RESEARCH CORP.
CASHOUR, JOHN	DEPARTMENT OF DEFENSE
CAUGHEL, BRIAN	CADRE TECHNOLOGIES
CERNOSEK, GARY	MCDONNELL DOUGLAS ASTRONANTICS CO.
CHANG, JOAN	COMPUTER SCIENCE CORP.
CHASSON, MARGARET	IBM CORP.
CHEADLE, BILL	MARTIN MARIETTA CORPORATION
CHEN, JENNIFER	COMPUTER SCIENCES CORP.

CHENN, PETER	UNIVERSITY OF MARYLAND
CHERNOFF, DARLENE	COMPUTER SCIENCES CORP.
CHESTER, ROWENA	MARTIN MARIETTA ENERGY SYSTEMS
CHIANG, TED	FORD AEROSPACE
CHILDERS, TIMOTHY	MARTIN MARIETTA
CHU, RICHARD	FORD AEROSPACE CO.
CHUNG, ANDREW	FAA TECHNICAL CENTER
CHURCH, VIC	COMPUTER SCIENCES CORP.
CISNEY, LEE	NASA/GSFC
CLARK, DAVID	UNISYS CORP.
CLIFTON, CHUCK	COMPUTER SCIENCES CORP.
COLAIZZI, DONALD	COMPUTER SCIENCES CORP.
COOK, JOHN	NASA/GSFC
COUCHOUD, CARL	SOCIAL SECURITY ADMINISTRATION
COURT, TERRY	HUGHES AIRCRAFT COMPANY
CRAIG, CLYDE	AUTOMETRIC, INC.
CREECY, RODNEY	HUGHES
CREWS, TERRY	LMSC
CRONE, MICHAEL	HARRIS CORP.
CROSS, JAMES	AUBURN UNIVERSITY
CUESTA, ERNESTO	COMPUTER SCIENCES CORP.
CUPAK, JOHN	HRB SYSTEMS
CURRY, DAN	MITRE CORP.
D'AGOSTINO, JEFF	OAO CORP.
DASKALANTONAKIS, MICHAEL	
DAVIS, CHARLES	TRW
DECKER, WILLIAM	COMPUTER SCIENCES CORP.
DELIS, ALEX	UNIVERSITY OF MARYLAND
DEUTSCH, MICHAEL	HUGHES AIRCRAFT CO.
DIXON, BERNARD	NASA/GSFC
DORBAND, JOHN	NASA/GSFC
DREW, DAN	UNISYS CORP.
DUNIHO, MICKEY	NSA
DUNN, NEPOLIA	COMPUTER SCIENCES CORP.
DUQUETTE, RICHARD	
DUREK, TOM	SPC, INC.
DUVALL, LORRAINE	DUVALL COMPUTER TECHNOLOGIES, INC.
DVONG, VINNIE	NSWC
DYER, MICHAEL	IBM/FSD
EBERHART, HERB	
EDELSTEIN, E.	GRUMMAN DATA SYSTEMS
EDGAR, ERIC	HRB — SYSTEMS
EGGERTSEN, KARL	NAVAL SHIPS WEAPONS SYSTEM ENG. STATION
EISENHARDT, GEORGE	LOGICON, INC.
ELLIS, WALTER	IBM
ELMORE, RALPH	COMPUTER SCIENCES CORP.
EMERY, KATHLEEN	VITRO CORP.
ENG, EUNICE	NASA/GSFC
ESKER, LINDA	COMPUTER SCIENCES CORP.
EVANCO, WILLIAM	THE MITRE CORP.
EVERS, JAY	UNISYS CORP.
FANG, HSIN	IBM
FANTASIA, DANIELE	UNIVERSITY OF MARYLAND

FELVER, HENRY	IBM
FERGUSON, FRANCES	STANFORD TELECOMMUNICATIONS
FESHAMI, BARBARA	SRA CORP.
FINK, MARY LOUISE	PLANNING RESEARCH CORP.
FINNEGAN, KENNETH	MARTIN MARIETTA CORP.
FORMANEK, KATHLEEN	MARTIN MARIETTA
FORSYTHE, RON	NASA/WALLOPS FLIGHT FACILITY
FOX, STEPHEN	XEROX ADVANCED INFORMATION TECHNOLOGY
FRANKLIN, JUDE	EMHART/PRC
FRANKS, KELLY	NASA/GSFC
GACUK, PETER	SPAR AEROSPACE
GAFFKE, WILLIAM	PROJECT ENGINEERING, INC.
GAFFNEY, JOHN	SPC, INC.
GANNETT, MARYE	DEPARTMENT OF DEFENSE
GARCIA, ENRIQUE	JET PROPULSION LAB
GARDNER, MICHAEL	COMPUTER SCIENCES CORP.
GIBSON, JOHN	IBM/SID
GILLILAND, DENISE	STANFORD TELECOMMUNICATIONS INC.
GILYEAT, COLIN	ADVANCED TECHNOLOGY, INC.
GIRONE, CHUCK	GE ASTRO SPACE
GODFREY, PARKE	UNIVERSITY OF MARYLAND
GODFREY, SALLY	NASA/GSFC
GOETTSCHE, CRAIG	NASA/GSFC
GOGIA, B.	ENGINEERING & ECONOMY RESEARCH
GOLDEN, JOHN	EASTMAN KODAK CO.
GOLDSMITH, LARRY	DEPT. OF LABOR
GOODSON, ADOLPH	NASA/GSFC
GORDON, HAYDEN	COMPUTER SCIENCES CORP.
GRAFTON, ED	LINK FLIGHT SIMULATION CORP.
GRAVES, RUSELL	DEPT. OF DEFENSE
GRAVITTE, JUNE	FORD AEROSPACE CORP.
GREEN, DANIEL	DOD
GREEN, SCOTT	NASA/GSFC
GREENBERG, DIANA	PRC
GREGORY, SAMUEL	
GRIMES, DONNA	IITRI
GRONDALSKI, JEAN	COMPUTER SCIENCES CORP.
GROSS, STEPHEN	NAVAL CENTER FOR COST ANALYSIS
HALL, GARDINER	FORD AEROSPACE CORP.
HALL, JAMES	UNISYS CORP.
HANEY, MODENNA	MARTIN MARIETTA
HARRIS, AL	LOGICAN, INC.
HARRIS, BERNARD	NASA/GSFC
HARTLEY, JONATHAN	NASA/GSFC
HASSETT, KEVIN	COMPUTER SCIENCES CORP.
HEASTY, RICHARD	COMPUTER SCIENCES CORP.
HEBENSTREIT, KARL	LOGICAN, INC.
HECK, JOANN	RMS TECHNOLOGIES, INC.
HEFFERNAN, HENRY	GCN
HEILIG, VICKI	IBM
HELLER, GERRY	COMPUTER SCIENCES CORP.
HENRY-NICKENS, STEPHANIE	NASA/GSFC
HENSON, TROY	IBM CORP.

HEYLIGER, GEORGE	COMPUTER TECHNOLOGY ASSOCIATES
HIHN, JAIRUS	JET PROPULSION LAB
HILDENBERGER, RUTH	MITRE CORP.
HILL, DONNA	NSWC
HODGES, WILLIAM	BOEING AEROSPACE CO.
HOLLORAN, PATRICK	SEI
HOLMES, BARBARA	CRMI
HOLOUBEK, DAN	LMSC
HOQ, N.	ENGINEERING & ECONOMY RESEARCH
HOUSER, WALTER	VETERANS ADMINISTRATION
HOUSTON, FRANK	FOOD & DRUG ADMIN.
HOWLE, BILL	NASA/MSFC
HUBER, HARTMUT	NSWC
HULL, LARRY	NASA/GSFC
HUTCHISON, ROBERTA	THE MITRE CORP.
JACKSON, LAVERNE	PLANNING RESEARCH CORP.
JAHANGIRI, MAJID	COMPUTER SCIENCES CORP.
JAKAITIS, JOYCE	AMERICAN SYSTEMS CORPORATION
JAWORSKI, ALLAN	SOFTWARE PRODUCTIVITY CONSORTIUM
JELETIC, JIM	NASA/GSFC
JESSEN, WILLIAM	RCA — ESD
JOESTING, DAVID	BENDIX FIELD ENGINEERING CORP.
JOHNSON, DONNA	COMPUTER SCIENCES CORP.
JOHNSON, RON	COMPUTER SCIENCES CORP.
JORDAN, LEON	COMPUTER SCIENCES CORP.
JUDKINS, HENRY	ATLIS FEDERAL SERVICES
KANG, KYO	SOFTWARE ENGINEERING INSTITUTE
KANNAPPAN, SAM	ABI ENTERPRISES
KAPLAN, STEVEN	DEPT. OF DEFENSE
KASCHAK, PAUL	NAVAL CENTER FOR COST ANALYSIS
KELLY, JOHN	JET PROPULSION LAB
KELLY, LISA	NASA/GSFC
KEMP, KATHRYN	VITRO CORP.
KERNAN, KEVIN	RATIONAL
KESTER, RUSH	GTE GOVERNMENT SYSTEMS
KILSDUNK, THOMAS	DEPT. OF DEFENSE
KIM, CHRISTINE	COMPUTER SCIENCES CORP.
KIM, SEUNG	COMPUTER SCIENCES OORP.
KIRBY, JAMES	SOFTWARE PRODUCTIVITY CONSORTIUM
KIRK, DANIEL	NASA/GSFC
KLITSCH, GERALD	COMPUTER SCIENCES CORP.
KNIGHT, JOHN	SPC, INC.
KOUCHAKDJIAN, ARA	UNIVERSITY OF MARYLAND
KOWALCHACK, BONNIE	APPLIED PHYSICS LAB
KRAMER, NANCY	PRC
KRAUS, PAUL	COMPUTATIONAL ENGINEERING, INC.
KUBARYK, PETER	IITRI
KUDLINSKI, ROBERT	NASA/LANGLEY
KUMAR, V.	NASA/STX
KURIHARA, TOM	U.S. DEPT. OF TRANSPORTATION
LABAUGH, ROBERT	MARTIN MARIETTA AEROSPACE CORP.
LAL, NAND	NASA/GSFC
LANDIS, LINDA	COMPUTER SCIENCES CORP.

LASKY, JEFFREY	ROCHESTER INSTITUTE OF TECHNOLOGY
LAVALLEE, DAVID	FORD AEROSPACE & COMM. CORP.
LAWRENCE-PFLEEZER, SHARI	GEORGE MASON UNIVERSITY
LEDFORD, RICK	MCDONNELL DOUGLAS CORP.
LEE, TOM	NASA/GSFC
LEENHOUTS, KATHLEEN	GENERAL ELECTRIC
LEFEVRE, JEANNE	UNISYS CORP.
LEFKOWITZ, SHARON	IIT RESEARCH INSTITUTE
LESAGE, LUCIAN	DEPT. OF DEFENSE
LIN, CHI	JPL
LINDSEY, JOEL	COMPUTER SCIENCES CORP.
LIU, JEAN	COMPUTER SCIENCES CORP.
LIU, KUEN-SAN	COMPUTER SCIENCES CORP.
LLOYD, MICHAEL	GENERAL DYNAMICS
LOESH, BOB	SYSTEM TECHNOLOGY INSTITUTE
LOWE, DAWN	NASA/GSFC
LUCZAK, EDWARD	COMPUTER SCIENCES CORP.
LUCZAK, RAY	COMPUTER SCIENCES CORP.
LYTTON, VICTOR	US DEPT OF AGRICULTURE
MACCHINI, BRUNO	UNIVERSITY OF MARYLAND
MACK, JOHN	ARCHITECTURE TECHNOLOGY
MALACANE, CHRISTINE	THE MITRE CORP.
MALHOTRA, SHAN	JPL
MANGIERI, MARK	JOHNSON SPACE CENTER
MANN, TIM	COMPUTATIONAL ENGINEERING, INC.
MARCINIAK, JOHN	MARCINIAK & ASSOCIATES
MARESCA, PAUL	ADASOFT, INC.
MARKUS, CYNTHIA	AMERICAN SYSTEMS CORP.
MARTIN, GEORGE	PROJECT ENGINEERING INC.
MASTER, PAT	IIT RESEARCH INSTITUTE
MATHIASEN, CANDY	UNISYS CORP.
MATTI, RUTH	NASA/GSFC
MAURY, JESSE	PLANNING & ANALYSIS CORP.
MCCONNAUGHERY, ED	DEPT. OF DEFENSE
MCDONALD, BETH	NASA/GSFC
MCGARRY, FRANK	GENERAL ELECTRIC
MCGARRY, PETER	HRB — SYSTEMS
MCKEAG, THOMAS	NSA
MCKENNA, JOHN	JET PROPULSION LAB
MCLEOD, JOHN	NASA/GSFC
MCQUILLAN, ARIEL	INSTITUTE FOR DEFENSE ANALYSIS
MEESON, REG	ADVANCED TECHNOLOGY
MERIFIELD, JAMES	COMPUTER SCIENCES CORP.
MILLER, JOHN	GENERAL ELECTRIC
MIRSCH, CYNTHIA	EER TECHNOLOGIES
MITTAL, AJAY	QSOFT, INC.
MOHANTY, SIBA	MARTIN MARIETTA ATC
MOHRMAN, CARL	CRMI
MOLESKI, LAURA	NASA/GSFC
MOLESKI, WALT	JET PROPULSION LAB
MOLKO, PATRICIA	MCDONNELL DOUGLAS ASTRONAUTICS CO.
MONTOYA, MARIA	CTA, INC.
MOORE, MIKE	WESTINGHOUSE ELECTRIC
MOORSHEAD, ART	

MOWERY, ED	THE MITRE CORP.
MOYLAN, ALDEN	COMPUTER SCIENCES CORP.
MRENAK, GARY	
MUDRONE, JAMES	DEPT. OF DEFENSE
MULARZ, DIANE	THE MITRE CORP.
MURPHY, ROBERT	NASA/GSFC
MUSA, JOHN	AT&T BELL LABORATORIES
MYERS, LEANNA	U.S. BUREAU OF LABOR STATISTICS
MYERS, PHILIP	COMPUTER SCIENCES CORP.
NARROW, BERNIE	BENDIX
NG, EDWARD	JET PROPULSION LAB
NGUYEN, BAO	HQ UASF/SCTT
NORCIO, TONY	UNIVERSITY OF MARYLAND
NORO, MASAMI	UNIVERSITY OF MARYLAND
O'BRIEN, ROBERT	NASA/GSFC
O'HARA-SCHETTINO, LIZ	GEORGE MASON UNIVERSITY
O'NEILL, L.	AT&T BELL LABS
OHLMACHER, JANE	SOCIAL SECURITY ADM.
OWENS, AL	NAVAL RESEARCH LAB
OWENS, KEVIN	PLANNING RESEARCH CORP.
PAGE, JERRY	COMPUTER SCIENCES CORP.
PAJERSKI, ROSE	NASA/GSFC
PALMER, JAMES	APL
PARKER, JAMES	IBM
PATEL, KANT	COMPUTER SCIENCES CORP.
PEARSON, BOYD	NASA/GSFC
PERKINS, DOROTHY	NASA/GSFC
PHILLIPS, GAIL	COMPUTER SCIENCES CORP.
PIETRASANTA, AL	
PIETSCH, ILA	DEPT. OF DEFENSE
PINCOSY, JOHN	DATA SYSTEMS ANALYSIS
PIXTON, JERRY	UNISYS CORPORATION
PLETT, MICHAEL	COMPUTER SCIENCES CORP.
PLUNKETT, THERESA	DEPT. OF DEFENSE
POLLACK, JAY	COMPUTER SCIENCES CORP.
POW, WILLIAM	
PRINCE, ANDY	PRS SYSTEMS SERVICES
PUGH, DOUGLAS	IIT RESEARCH INSTITUTE/DQT
PUMPHREY, KAREN	COMPUTER SCIENCES CORP.
PUTNEY, BARBARA	NASA/GSFC
QUANN, EILEEN	FASTRAK
QUIMBY, KELVIN	COMPUTER SCIENCES CORP.
RACINE, GLENN	AIRMICS
RANSOM, BERT	NASA/GSFC
RASH, JAMES	NASA/GSFC
RAUTNER, JIM	MOUNTAINET, INC.
RAWERS, KEVIN	LOCKHEED
REEDY, CHRISTOPHER	BETAC CORP.
RICE, RAYMOND	MCDONNELL DOUGLAS ASTRONAUTICS, CO.
RIGNEY, BRANDON	PRC
RITTER, SHEILA	NASA/GSFC
ROBBINS, DON	GTE — GOVERNMENT SYSTEMS
ROBERTS, BECKY	PRC
ROBINSON, MARY	THE MITRE CORP.

ROBINSON, RICHARD	THE MITRE CORPORATION
ROBINSON, STEVE	DYNAMICS RESEARCH CORP.
ROBISON III, W.	JET PROPULSION LAB
ROGERS, KATHY	THE MITRE CORP.
ROHR, JOHN	JET PROPULSION LAB
ROMBACH, DIETER	UNIVERSITY OF MARYLAND
ROSS, DON	IIT RESEARCH INSTITUTE
ROUNDS, CHUCK	SRA CORP.
ROY, DANIEL	FORD AEROSPACE CORP.
RUCKI, DAN	DEPT. OF DEFENSE
RUPERT, FRED	FEDERAL HOME LOAN MORTGAGE CORP.
RUTEMILLER, OREN	STANFORD TELECOMMUNICATIONS, INC.
SABIA, STEVE	NASA/GSFC
SABOTIN, ROSA	COMPUTER SCIENCES CORP.
SALOMON, ARTHUR	STANFORD TELECOMMUNICATIONS, INC.
SAMSON, DOLLY	GEORGE MASON UNIVERSITY
SAYANI, HASAN	ADVANCED SYSTEMS TECH CORPORATION
SCAVETTI, JOSEPH	AMERICAN SYSTEMS CORP.
SCHUBERT, KATHY	NASA/LERC
SCHULTHEISZ, ROBERT	NATIONAL LIBRARY OF MEDICINE
SCHWARTZ, MICHAEL	IITRI/ECAC
SCHWENK, ROBERT	NASA/GSFC
SCIULLO, ED	NATIONAL LIBRARY OF MEDICINE
SCOTT, LEIGHTON	NSA
SEAVER, DAVID	PROJECT ENGINEERING, INC.
SEIDEWITZ, ED	NASA/GSFC
SEIGLE, JEFF	COMPUTER SCIENCES CORP.
SELVAGE, ROB	U.S. TREASURY/FMS
SEVER, GEORGE	MARTIN MARIETTA
SEVERINO, TONY	GENERAL ELECTRIC/RCA
SHANK, DWIGHT	COMPUTER SCIENCES CORP.
SHEN, VINCENT	MCC
SHEPPARD, SYLVIA	COMPUTER TECHNOLOGY ASSOCIATES
SHERE, KEN	AVTEC SYSTEMS
SHI, LEON	COMPUTER SCIENCES CORP.
SHOAN, WENDY	NASA/GSFC
SHUPE, GARY	NAVAL DATA AUTOMATION COMMAND
SHUSTER, DAVID	DATA SYSTEMS ANALYSTS
SHYMAN, STEVEN	INSTITUTE FOR DEFENSE ANALYSES
SIEG-ROSS, SANDY	U.S. EPA
SINCLAIR, SEAN	COMPUTER SCIENCES CORP.
SKINNER, JUDITH	JET PROPULSION LAB
SMITH, DAN	FORD AEROSPACE CORP.
SMITH, KATHRYN	NASA/LARC
SMITH, LEN	COMPUTER SCIENCES CORP.
SMITH, PATRICIA	NSWC
SNYDER, TIM	COMPUTER SCIENCES CORP.
SOLOMAN, CARL	NASA/GSFC
SOVA, DONALD	NASA/HQ
SPANGLER, ALAN	IBM
SPENCE, BAILEY	COMPUTER SCIENCES CORP.
SPIEGEL, DOUG	NASA/GSFC
SPIEGEL, MITCHELL	GTE SYSTEMS
SRIRANGARAJAN, RAJAN	THE MITRE CORP.

STANLEY, CAROLYN	MARTIN MARIETTA
STARK, MICHAEL	NASA/GSFC
STEINBACHER, JODY	JET PROPULSION LAB
STEINBERG, SANDEE	COMPUTER SCIENCES CORP.
STEVENSON, JEFF	MARTIN MARIETTA
STEWART, CHARLES	CRMI
STOKES, SAM	THE MITRE CORP.
STRAUB, PABLO	UNIVERSITY OF MARYLAND
STUMPO, PAUL	DEPT. OF DEFENSE
SUBOTIN, ROSA	COMPUTER SCIENCES CORP.
SUD, VED	MITRE CORP.
SUN, ALICE	THE MITRE CORP.
SWAIN, BARBARA	UNIVERSITY OF MARYLAND
SYMMES, BRIAN	U.S. EPA
SZULEWSKI, PAUL	C.S. DRAPER LABS, INC.
TASAKI, KEIJI	NASA/GSFC
TAUSWORTHE, ROBERT	JPL
TAYLOR, TOM	BUREAU OF THE CENSUS
THACKERY, KENT	PLANNING ANALYSIS CORP.
THEOFANOS, MARY	MARTIN MARIETTA ENERGY SYSTEMS
THOMPSON, JIM	FREDDIE MAC
THOMPSON, JOHN	FORD AEROSPACE
THOMPSON, WILLIAM	NSWC
THORNTON, THOMAS	JPL
THRASYBULE, WESNER	COMPUTER SCIENCES CORP.
TOMPKINS, JEFF	COMPUTER SCIENCES CORP.
TRAN, LAN	JET PROPULSION LAB.
TREFFER, LEIGH	IITRI
TSOUNOS, ANDREW	SEI
ULERY, BRADFORD	UNIVERSITY OF MARYLAND
USAVAGE, PAUL	GENERAL ELECTRIC
VALETT, JON	NASA/GSFC
VALETT, SUSAN	NASA/GSFC
VAN DITTA, MARK	INFORMATION SYSTEMS & NETWORKS CORP.
VERNACCHIO, AL	NASA/GSFC
VOGEL, MICHAEL	PRC
WALIGORA, SHARON	COMPUTER SCIENCES CORP.
WALKER, GARY	JET PROPULSION LAB
WALLACE, CHARLES	RAYTHEON SERVICE CO.
WALTMAN, ROBERT	IBM
WATERMAN, BOB	VITRO CORP.
WATSON, STAN	NASA/GSFC
WEBSTER, THOMAS	COMPUTATIONAL ENGINEERING INC.
WEISMAN, DAVID	UNISYS CORP.
WEISS, DAVE	SPC, INC.
WELBORN, RICHARD	STANFORD TELECOMMUNICATIONS, INC.
WELLS, CYNTHIA	COMPUTATIONAL ENGINEERING, INC.
WENDE, CHARLES	NASA/GSFC
WHEELER, JIM	NAVAL DATA AUTOMATION COMMAND
WILLIAMSON, DAVID	IITRI
WILLIAMSON, PHIL	BOEING COMPUTER SUPPORT SERVICES
WILSON, JEAN	MDAC/KSC
WONG, WILLIAM	NATIONAL INSTITUTE OF STANDARDS & TECH.
WOOD, RICHARD	COMPUTER SCIENCES CORP.

WOOD, TERRI	NASA/GSFC
WRIGHT, CYNTHIA	THE MITRE CORP.
WU, SABINA	IITRI
WU, YEN	IITRI
YAAKOV, BEN-AMI	JET PROPULSION LAB
YANG, CHAO	NASA/GSFC
YEE, MARY	LOGICON, INC.
YENCHZ, MARTIN	WESTINGHOUSE
YU, STELLA	COMPUTER SCIENCES CORP.
YUNG, K	COMPUTER SCIENCES CORP.
ZAVELER, SAUL	US AIR FORCE
ZELKOWITZ, MARV	UNIVERSITY OF MARYLAND
ZIMET, BETH	COMPUTER SCIENCE CORP.
ZYGIELBAUM, ART	JET PROPULSION LAB

APPENDIX B

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-004, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-706, Annotated Bibliography of Software Engineering Laboratory Literature, G. Heller, January 1989

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From the Tenth Annual Software Engineering Workshop, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986

SEL-86-005, Measuring Software Design, D. N. Card, October 1986

SEL-86-006, Proceedings From the Eleventh Annual Software Engineering Workshop, December 1986

SEL-87-001, Product Assurance Policies and Procedures for Flight Dynamics Software Development, S. Perry et al., March 1987

SEL-87-002, Ada Style Guide (Version 1.1), E. Seidewitz et al., May 1987

SEL-87-003, Guidelines for Applying the Composite Specification Model (CSM), W. W. Agresti, June 1987

SEL-87-004, Assessing the Ada Design Process and Its Implications: A Case Study, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-008, Data Collection Procedures for the Rehosted SEL Database, G. Heller, October 1987

SEL-87-009, Collected Software Engineering Papers: Volume V, S. DeLong, November 1987

SEL-87-010, Proceedings From the Twelfth Annual Software Engineering Workshop, December 1987

SEL-88-001, System Testing of a Production Ada Project: The GRODY Study, J. Seigle and Y. Shi, November 1988

SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988

SEL-88-003, Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis, K. Quimby and L. Esker, December 1988

SEL-RELATED LITERATURE

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[4]Basili, V. R., and D. Patnaik, A Study on Fault Prediction and Reliability Assessment in the SEL Environment, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

Basili, V. R., and J. Ramsey, Structural Coverage of Functional Testing, University of Maryland, Technical Report TR-1442, September 1984

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE Computer Society Press, 1979

[5]Basili, V. and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," Proceedings of the 9th International Conference on Software Engineering, March 1987

[5]Basili, V. and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," Proceedings of the Joint Ada Conference, March 1987

[5]Basili, V. and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, June 1988

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland, Technical Report TR-1501, May 1985

[3]Basili, V. R. and R. W. Selby "Four Applications of a Software Data Collection and Analysis Methodology," Proceedings of the NATO Advanced Study Institute, August 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

[5]Basili, V. and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, December 1987

[2]Basili, V. R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: IEEE Computer Society Press, 1978

[5]Brophy, C., W. Agresti, and V. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," Proceedings of the Joint Ada Conference, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," Proceedings of the Washington Ada Technical Conference, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[5]Card, D. and W. Agresti, "Resolving the Software Science Anomaly," The Journal of Systems and Software, 1987

[6]Card, D. N., and W. Agresti, "Measuring Software Design Complexity," The Journal of Systems and Software, June 1988

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D., F. McGarry, and G. Page, "Evaluating Software Engineering Technologies," IEEE Transactions on Software Engineering, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," ACM Software Engineering Notes, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: IEEE Computer Society Press, 1983

[5]Doubleday, D., "ASAP: An Ada Static Source Code Analyzer Program," University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S. and C. Brophy, "Experiences in the Implementation of a Large Ada Project," Proceedings of the 1988 Washington Ada Symposium, June 1988

Hamilton, M., and S. Zeldin, A Demonstration of AXES for NAVPAK, Higher Order Software, Inc., TR-9, September 1977 (also designated SEL-77-005)

Jeffery, D. R., and V. Basili, "Characterizing Resource Data: A Model for Logical Association of Software Data," University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," Proceedings of the Tenth International Conference on Software Engineering, April 1988

[5]Mark, L. and H. D. Rombach, "A Meta Information Base for Software Engineering," University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L. and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[5]McGarry, F. and W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (Proceedings), March 1980

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[5]Ramsey, C. and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," University of Maryland, Technical Report TR-1708, September 1986

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," IEEE Transactions on Software Engineering, March 1987

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," Proceedings From the Conference on Software Maintenance, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," Proceedings of the 21st Hawaii International Conference on System Sciences, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," Proceedings of the CASE Technology Conference, April 1988

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1987

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," Proceedings of the Joint Ada Conference, March 1987

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[5]Valett, J. and F. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[5]Wu, L., V. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," Proceedings of the Joint Ada Conference, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," Journal of Systems and Software, 1988

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

NOTES:

[1]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[4]This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.

[5]This article also appears in SEL-87-009, Collected Software Engineering Papers: Volume V, November 1987.

[6]This article also appears in SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988.